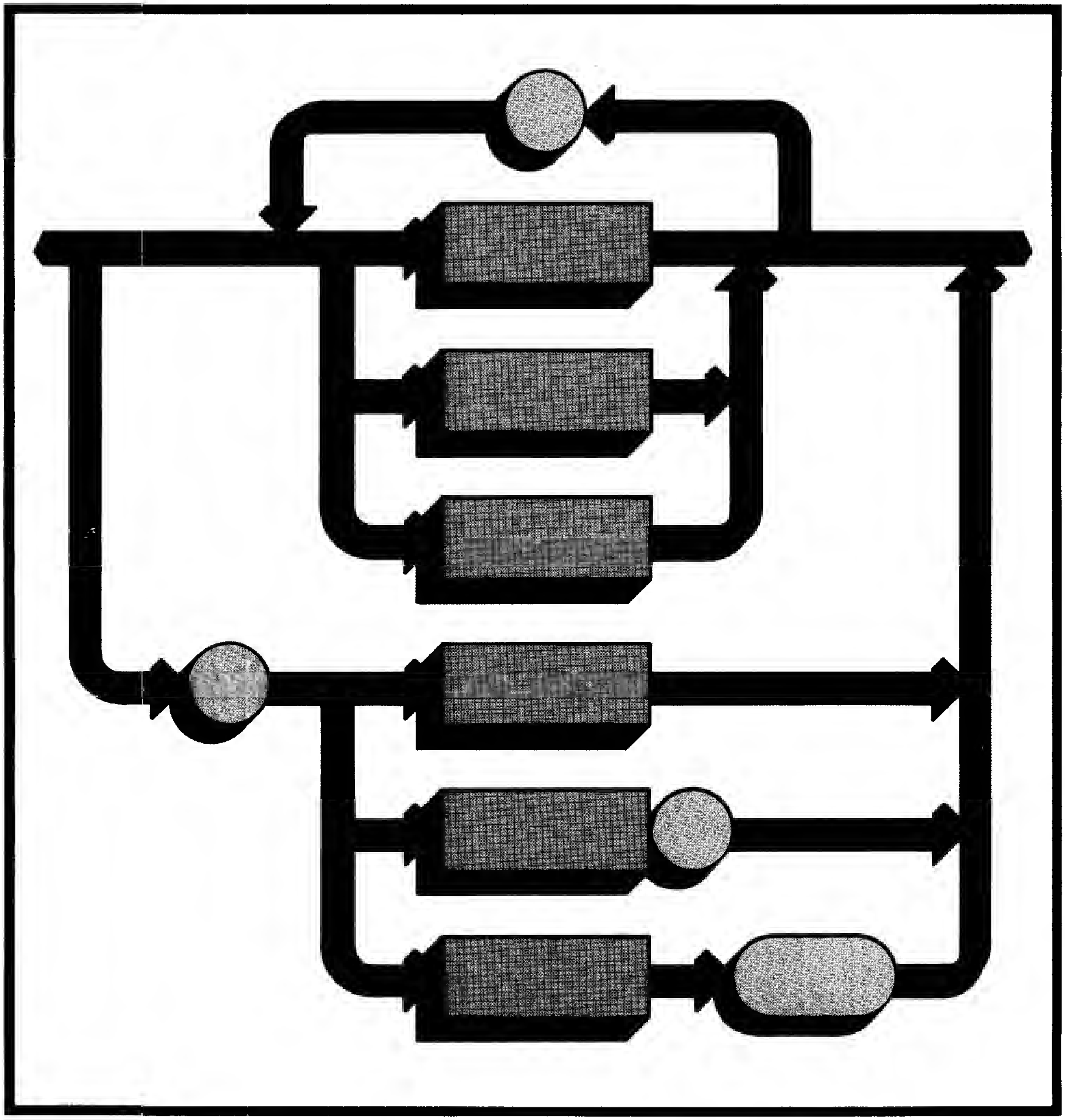
HP 9000 Series 200 Computers

HEWLETT PACKARD

Pascal 3.0 Workstation System



Pascal 3.0 Workstation System

for the HP 9000 Series 200 Computers

Manual Part No. 98615-90021

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

May 1984...First Edition

Warranty Statement

Hewlett-Packard products are warranted against defects in materials and workmanship. For Hewlett-Packard Fort Collins Systems Division products sold in the U.S.A. and Canada, this warranty applies for ninety (90) days from the date of delivery.* Hewlett-Packard will, at its option, repair or replace equipment which proves to be defective during the warranty period This warranty includes labor, parts, and surface travel costs, if any. Equipment returned to Hewlett-Packard for repair must be shipped freight prepaid. Repairs necessitated by misuse of the equipment, or by hardware, software, or interfacing not provided by Hewlett-Packard are not covered by this warranty.

HP warrants that its software and firmware designated by HP for use with a CPU will execute its programming instructions when properly installed on that CPU. HP does not warrant that the operation of the CPU, software, or firmware will be uninterrupted or error free.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

* For other countries, contact your local Sales and Support Office to determine warranty terms

Table of Contents

Chapter 1: Overview

Chapter 3: The Editor

Chapter 4: The Filer

Chapter 5: Pascal Compiler

Chapter 6: The Assembler

Chapter 7: The Librarian

Chapter 8: The Debugger

Overview

Chapter 1

Introduction

This manual describes using the HP Series 200 Pascal 3.0 Workstation System. It focuses on how to use the subsystems of the Pascal Workstation "environment" – the Editor, Filer, Compiler, Assembler, Librarian, and Debugger – and how they interact to provide you with a powerful Pascal program development tool.

Before Reading this Manual

Here are the manuals that you should have read *before* reading this manual.

Documentation Guide

The *Pascal 3.0 Documentation Guide* describes each manual in the documentation set. It will help you to learn where the various parts of the system are described.

Installation Guide

You should have already set up your computer hardware according to the instructions in the *Installation Guide* for your computer. If not, you should do that now.

User's Guide

You should have also booted the Pascal system according to the instructions in the *Pascal 3.0 User's Guide*. You should have also followed along with the examples to learn how to begin using the system to compile a few simple Pascal programs.

Note

This manual describes many keyboard operations. Since you can have one of three different keyboards, this manual generally describes the keystrokes required on all three keyboards.

For instance, on the HP 46020A keyboards, there are both (Enter) and (Return) keys, while on the HP 98203A and 98203B keyboards, there is only an (ENTER) key. When you are directed in this manual to press one of these keys, the text will usually say: "Press the (Return) or (Enter) key."

Another common example is the (Select) key on the 46020 and the (EXECUTE) key on the 98203 keyboards. When you are directed to press one of these keys, the text will say: "Press the (Select) ((EXECUTE)) key." (The second key noted in parentheses is the 98203 key.)

Descriptions of each keyboard and key-correspondence tables are presented in the *Pascal 3.0 User's Guide*.

Pascal Textbook

The system manual (the one you are now reading) fully describes the tasks of entering, editing, storing, compiling, loading and executing, and debugging Pascal programs. However, the manual does **not** contain all of the "programming techniques" information you will need to fully exploit the power of the Pascal language. Thus, if you are not familiar with the Pascal language, you should read *An Introduction to Programming and Problem Solving with Pascal* (included in the manual set sent with your system).

Other Series 200 Manuals

This manual does not generally assume that you are familiar with any of the other languages and systems available for Series 200 computers, although references are occasionally made to some of these other languages where appropriate (such as BASIC).

Previous Workstation Pascal Manuals

If you are familiar with the documentation for earlier versions of the Pascal Workstation System, you may be happy to know that this manual is a later edition of the *Pascal User's Manual.* However, this manual describes only Version 3.0 of the Pascal Workstation system. The main text does not generally discuss earier versions of the system.

If you are upgrading from an earlier version, you may first want to read the System History section in the Technical Reference appendix of this manual (the one you are now reading).

Chapter Previews

Here are brief previews of the contents of each of the chapters of this manual.

Chapter 1: Overview

The remainder of this chapter describes the commands available at the Main Command Level.

Chapter 2: The File System

This chapter introduces you to the Workstation File System. It describes how the logical units and volumes are organized and accessed, and it also provides an outline of programming with files.

Chapter 3: The Editor

A program usually starts out as an idea. The Editor's function is to provide a useful environment for the translation of thoughts into actual programs or documents. This chapter fully explains the features of the Pascal Workstation Editor.

Chapter 4: The Filer

The Filer is used to store, load, copy, translate and perform other file-related utility operations. This chapter details performing these operations with the Filer.

Chapter 5: The Pascal Compiler

Once a program has been written with the Editor, this source code must be compiled into object code before it can be executed. This chapter explains the operation of the Compiler and the options that can be used to modify its operation. The chapter also describes the modular programming capability, which is one of the most powerful features of this system.

Chapter 6: The Assembler

This chapter introduces you to the Assembler, which converts programs written in MC68000 Assembler language – a humanly understandable version of the microprocessor's machine language – into object code for the MC68000 family of processors used in the Series 200 Computers.

Chapter 7: The Librarian

This chapter covers using the Librarian. In the system are libraries of object-code modules: some consist of device-drivers, while others consist of useful procedures for such applications as I/O and graphics. You can also design your own modules. The Librarian's function is to manage libraries of Pascal and Assembler language object modules.

Chapter 8: The Debugger

We all wish that a program would run perfectly the first time. Unfortunately, there is little evidence in real life to support that fantasy. The next best thing is to have some good tools to help you debug your programs. This chapter explains the debugging features available with this system.

Chapter 9: Special Configurations

This chapter describes how to set up "non-standard" configurations. It first gives background information regarding how the system boots and configures itself, and then it describes the steps required to set up several configurations.

Chapter 10: Non-Disc Mass Storage

Several "non-disc" types of mass storage devices are available on the Series 200 Pascal Workstation: EPROM (Erasable Programmable Read-Only Memory) cards, Magnetic Bubble Memory cards, and DC600 tape cartridge drives. Configuring and using these devices is described in this chapter.

Technical Reference Appendix

This appendix contains the following information:

- A history of the Pascal system, which includes descriptions of the differences between the 3.0 and previous versions of the Workstation system
- A list of module names used by the 3.0 system
- Software memory map
- ASCII character tables

Command Summaries

This appendix contains a summary of commands for each of the Pascal subsystems.

Glossary

Knowing what technical terms mean is always useful.

Error Messages

This appendix contains the complete listings of all error messages for the various Pascal subsystems.

Index

This section contains an index to the topics in this manual.

The Main Command Level

The Main Command Level is the central point of reference for the operating system. It is "where you are" after booting the system and after each program completes. This section describes in detail those Main Command Level operations which do *not* call subsystems (such as the Editor, Filer, Compiler, etc.); the subsystems are each described in later chapters of this manual. However, all the Main Command Level commands are listed in the subsequent Quick Reference.

Main Command Prompt

The Main Command Level consists of two prompt lines, only one of which is displayed at one time. Press the ? key to toggle between them.

```
Command: Compiler Filer Editor Initialize Librarian Run eXecute Version ?

Command: Assembler Debugger Memvol Newsysvol Permanent Stream User What ?
```

The uppercase letters in the prompt lines indicate which key to press to start the operation.

All of the operations are available regardless of which prompt is being displayed.

The prompts are abbreviated on the 50 column display of the Model 226.

```
Command: Cmplr Edit File Init Libr Run Xcut Ver ?

Command: Asm Dbg Memv New Perm Stream User What ?
```

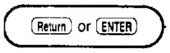
Main Command Quick Reference

Command	Description
Compiler	Calls the Compiler to translate Pascal source code into object code.
Editor	Calls the Editor for creating or editing a source program or textual document.
Filer	Calls the Filer for management of the File System.
Initialize	Initializes the File System (but not discs).
Librarian	Calls the Librarian for managing, linking, or unassembling object-code files.
Run	Runs the workfile (compiling it if needed) or the last program compiled since power-up. If there is no workfile, Run operates like eXecute.
eXecute	Asks for a code file and runs it.
Version	Allows setting the time and date, and displays all the current system version information.
Assembler	Calls the Assembler to translate an Assembler language source program into object code.
Debugger	Runs a program under control of the Debugger.
Memory volume	Sets up a memory resident mass storage volume for fast access.
New sysvol	Asks for a volume to be designated as the system volume.
Permanent	Asks for a code file to be permanently loaded into memory for execution without disc loading each time.
Stream	Asks for a stream file whose characters are interpreted as keyboard input until there are no more left.
User restart	Restarts the last program or subsystem that was run.
What	Displays the system file table and allows you to change the system files or system and default volumes.

Main Command Reference

Each command in this section contains a description and a syntax diagram. The syntax diagrams contain rounded and rectangular boxes. Elements in rounded boxes should be interpreted as literals. An example is as follows:

This notation indicates that you must literally type a C as part of the command.

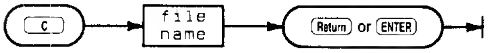


The Return or Enter indicates that you can press either key.

Elements in rectangular boxes are non-literal descriptions of command parameters. An example is as follows:

This notation indicates that you must supply the actual file name as part of the command.

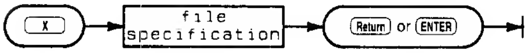
An example of a complete command is as follows:



If, for example, this was the Compiler command syntax diagram, it would mean that you must type C to run the Compiler, then type the name of the file to be compiled, and enter the file name with either the Return or the Enter key.

eXecute

The eXecute command runs a specified code file.



Item	Description/Default	Range Restrictions
file specification	literal	Any legal file specification (see the File System chapter)

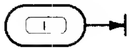
Semantics

The file you specify should be previously compiled or assembled and ready to run. It is not necessary to include the `.CODE` suffix in the file name; it is automatically appended to the file name if not included. If the actual file name does not contain a `.CODE` suffix, you will need to terminate the file specification with a period to suppress this suffix.

If the specified code file imports other modules, those modules must be contained in the file being executed, in the current System Library (which must be on-line), or they must be Permanently loaded (by using the Permanent command at the Main Level). You can use the What command to see which file is designated as the current System Library, and to change it if desired.

Initialize

The Initialize operation updates Unit Table entries for all units that are currently on-line. (It does not initialize mass storage media; that function is performed by using the MEDIAINIT utility program. See the *Pascal 3.0 User's Guide* for further details.)



Semantics

The Unit Table contains a record for each of 50 possible logical units available to the File System. The assignment of unit numbers to physical devices (auto-configuration) is performed by the TABLE program at power-up. Each record contains the "device address vector" of the physical device which corresponds to that logical unit number. The computer then looks at the physical location indicated by the device address vector to see if the device is on-line. If it is, that fact is marked in the record for that unit, along with the volume name (if media is currently installed in the device). Afterwards, the computer only looks at the Unit Table to see if a particular device is on-line; it does not check the actual device. (See the Booting Process section of the Special Configurations chapter for further details of how the TABLE program works.)

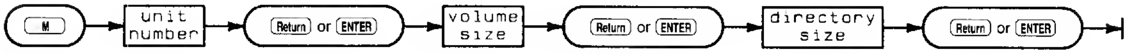
When a device is added to your system after the computer has been powered-up, you will usually need to execute BOOT:TABLE or power-up the system again in order for the device to be recognized. However, the Initialize command may in some cases be sufficient to get the system to recognize the new device.

Initialize also performs a device clear for all on-line devices and causes the system to forget the last loaded file (the User command can't reload the last program). The Initialize operation also causes all temporary files to be removed from each volume the next time a file is opened on the volume.

The volumes CONSOLE: (Unit #2) and PRINTER: (Unit #6) are special cases; these volumes are always assumed to be on-line. Thus, the system may "hang" if either of them is off-line.

Memory volume

The Memory volume command creates a mass storage volume in memory.



Item	Description/Default	Range Restrictions
unit number	integer	7 thru 50
volume size	integer indicating the number of 512-byte blocks	≥1
directory size	integer indicating the maximum number of files in the volume	≥1

Semantics

The Memvol command gives you the capability for very fast mass storage operations.

When the Memvol command is given, you are prompted for a unit number. This number corresponds to an entry in the Unit Table. Don't give a unit number which is already in use. The Volumes command in the Filer subsystem shows which unit numbers are currently used. For most applications 50 is the recommended unit number to use for your first memory volume.

You are then prompted for the number of (512-byte) blocks needed for the memory volume. Try to estimate conservatively the amount of memory you want reserved for the memory volume because it cannot be returned for general purpose use without turning off the computer. On the other hand, if you don't specify enough space, you have to create another larger volume.

Memory volumes are useful for program development where a lot of mass storage I/O (editing and compiling) is involved. Reserve enough space on the memory volume for both the source file, the object code file, and 40 extra blocks for the Compiler's temporary files. A good rule of thumb is:

size_of_volume = size_of_source_file (in 512-byte blocks) * 4 + 40

If you are transferring a source file from disc (as opposed to starting from scratch) you can determine its size by getting a directory listing of the volume that contains it. However, note that different directories return the file size in different units.

- LIF directories use 256-byte "blocks"
- WS1.0 directories use 512-byte "blocks"
- SRM directories use 1-byte "blocks"

Note that the default directory access method (DAM) for memory volumes is LIF; this DAM is the primary DAM specified in the TABLE program. See the Special Configurations chapter for further details about changing the primary DAM.

You are then prompted to give the number of directory entries you need for this memory volume.

```
Number of directory entries ?
```

Type the number you think you'll need and press Enter.

You can refer to your memory volume by it's unit number. For example:

```
#50:
```

Alternately, you can refer to it by its given volume name, which is initially RAM:. For example:

```
RAM:
```

If you plan to use more than one memory volume, use the Filer's Change command to give each memory volume a unique name.

Here is a method for setting up an extremely fast program development environment.

1. Create a RAM: volume and specify it as the system volume using the Newsysvol command.

   Specify RAM: as the default volume using the Main Command Level's What command or the Filer's Prefix command.
2. Permanently load the Editor and Compiler using the Permanent command.
3. Go into the Editor and write your program.
4. When you're ready to leave the Editor, use the Update option to create a workfile. The system puts the workfile on the fast RAM system volume.
5. Press R.

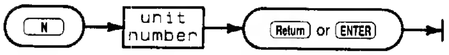
Your file will automatically be compiled. If it compiles with no errors, it will be run. If it contains errors, you will have the option of returning to the Editor.

Note

Since memory volumes are volatile, don't forget to save the files in the memory volume on a disc before turning off the computer.

New sysvol

The New sysvol command specifies a new system volume and updates the operating system file table accordingly.



Item	Description/Default	Range Restrictions
unit number	integer	1 thru 50

Semantics

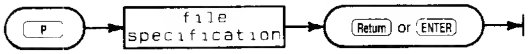
The system file table is used in locating operating system files. It contains the volume and file names of system files (EDITOR, FILER, etc.). When you press a key at the Main Command Level that invokes one of these subsystems (such as E), the system attempts to load the corresponding file indicated in the system file table (here, the EDITOR file).

You can use this command to specify a new system volume. The first step in this operation prompts you for a unit number. The device corresponding to the specified unit number is considered to be the new system volume, and serves as a starting point in the search for the system files: ASSEMBLER, COMPILER, EDITOR, FILER, LIBRARIAN, LIBRARY, and the work file. If any of these system files are not found, the Unit Table is used in a sequential search for the rest of them. As each file is found, the name of the volume on which it is found is prepended to the file name (for instance, SYSVOL:LIBRARY), and the complete file specification is placed in the file table. If any system file is not found in this search, the operating system assumes that it will find the file on the flexible disc volume on which it was delivered (for instance, ACCESS:EDITOR).

Use the Main Command Level's What command to see the resultant system file table.

Permanent

The Permanent command loads a program permanently into memory.



Item	Description/Default	Range Restrictions
file specification	literal	Any legal file specification (see the File System chapter)

Semantics

The Permanent command can be used to load a user program, a system program (Editor, Compiler etc.), or a module that is needed by a program. This code file is then ready to execute immediately when the command is given. A "P-loaded" (Permanently loaded) program does not have to be loaded from disc each time it is run.

After you give the Permanent command, you are prompted for the name of the file which contains the module or program. You need not include the `.CODE` suffix; if you don't include one, the suffix will be appended to the file name. If the file to be P-loaded does not have a `.CODE` suffix, end the file specification with a period to suppress the suffix from being appended to the file name automatically.

Several programs may be P-loaded in memory. The operating system keeps track of which programs have been P-loaded. When you give a command to run a program, the operating system checks to see if it has been P-loaded; if so, it is executed immediately. If not, it is loaded from disc and then executed; after execution, the memory used by the program is reclaimed.

An object module which is imported by a program must be in the object file that contains the program, in a file previously P-loaded, or contained in the current System Library (which must be on-line).

A program or module's global variables are zeroed only when it is loaded, not each time the program is run. However, note that neither local variables nor dynamic variables are zeroed.

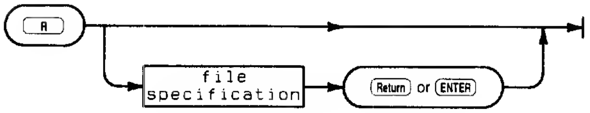
Note

Th volume name is not retained when a file is P-loaded. Attempting to execute a file of the same name but on a different volume will still result in the P-loaded file being executed.

For SRM users, do not use a directory path name to execute a P-loaded file.

Run

The Run operation causes the workfile or last compiled program to be executed.



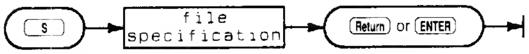
Item	Description/Default	Range Restrictions
file specification	literal	Any legal file specification (see the File System chapter)

Semantics

When the Run command is given, the operating system checks to see if there is a workfile. If there is a CODE workfile, it is executed; if not, the most recently compiled or assembled file is executed. If there is a TEXT workfile but no CODE workfile, the TEXT workfile is first compiled (with the system compiler) to a CODE file and then the CODE file is executed. If there is no workfile or previously compiled program, the command operates like the eXecute command and you are prompted for a file specification.

Stream

The Stream command "executes" a file of ASCII characters as if they were being typed from the keyboard.



Item	Description/Default	Range Restrictions
file specification	literal	Any legal file specification (see the File System chapter)

Semantics

A command stream or stream file is a file that is interpreted as input to the Main Command Level and/or its subsystems in place of keyboard input. The Stream operation causes a file to be interpreted. Therefore, a stream file is useful for executing a sequence of commonly used commands without requiring any operator intervention.

A stream file is created with the Editor and may be of type ".TEXT", ".ASC" or Data. If you do not specify a suffix, a ".TEXT" is automatically appended to the file name; if the name of the file to be streamed does not have a suffix, add a trailing period to the file name to suppress the suffix.

In order to generate a valid sequence of keystrokes, you should first run through the desired sequence while noting the keystrokes entered. Note particularly the occurences or absences of the Return or Enter key. Then enter the same keystrokes in your stream file. If, during an Editor or Filer command sequence, you encounter an unpredictable question that has a (Y/N) or (R/O/N) question associated with it, do not answer the question in the stream file. These kinds of questions are answered automatically as the file is streamed. (Y/N) questions (Yes/No) are answered "Y". (R/O/N) questions (Remove/Overwrite/Neither) are automatically answered "R".

After all the characters in a stream file have been interpreted, control is returned to the keyboard.

Comments

Stream files may contain comments. A line beginning with an asterisk (*) will be interpreted as a comment if it occurs at the Main Command Level. (Comments cannot be embedded among commands for subsystems or user programs.) When the command interpreter encounters one or more comment lines while streaming, they are displayed briefly on the screen, thus allowing the process to be monitored.

Immediate Execute Keys

If it is necessary to use keys that also act as immediate-execute commands in the Editor, such as Select (EXECUTE) or Backspace, use the following key sequences to generate those keystrokes.

Immediate-Execute Key	Generate with these Keys
Select	CTRL-Select C
Backspace	CTRL-Select H
Tab	CTRL-Select I
Clear display	CTRL-Select L
Left arrow	CTRL-Select H
Right arrow	CTRL-Select <
Up arrow	CTRL-Select ?
Down arrow	CTRL-Select J

If you have a 98203 keyboard, substitute EXECUTE or EXEC for Select in the preceding table.

Prompts for Keyboard Input

A stream file can be made to display a prompt on the CRT and then wait for an input string from the keyboard. The input string is assigned to a variable in the stream file. When the variable is encountered during streaming, the string is used in its place.

This input prompting must appear in the stream file *before all* of the commands or comments. Up to 36 prompts are allowed. They are denoted with an "=" as the first character on a line.

To prompt for an input string, place an equal sign, followed by a single alphanumeric character variable name (uppercase and lowercase letters used for variables are treated as equal), followed by the prompt text. For example:

```
=f What is the name of the file to be P-loaded ?
```

When the file name is typed in response to the prompt, it is stored in the specified variable, in this case the variable named f.

After the input prompting, begin entering the commands in the stream file. When you want the input string to be given to the operating system, use the variable preceded by "@". For example, the following characters are a command stream:

```
P@f
```

The command is the Permanent load command with a file name parameter indicating which file is to be P-loaded. The file whose name was given in response to the above prompt is then P-loaded.

Disabling the Prompt Feature

If the stream file name contains a [*] specifier, the ability to prompt for keyboard input is disabled.

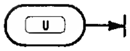
(Normally when a file is Streamed, the file is copied to the file named STREAM on the current system volume; during this copy, prompts are displayed and @ variables assigned values input from the keyboard by the computer operator. After all variables have been assigned, the file is read as keystrokes; in other words, *STREAM is the file that is actually streamed. The [*] suppresses the normal processing of the prompts and input variables, as the keys are read directly from the specified file.)

Stream Files on Read-Only Devices

It is the disabled-prompt mechanism (see preceding discussion) that allows the use of stream files stored on read-only mass storage, such as EPROM, and the use of read-only devices as system volumes. This mechanism is also used to process the AUTOKEYS stream file, if found during the boot process when the AUTOSTART stream file is not present. For examples of AUTOSTART and AUTOKEYS stream files, see the discussions in the *Pascal 3.0 User's Guide* and in the Special Configurations chapter of this manual.

User restart

The User restart command causes the last program that was run to be rerun.



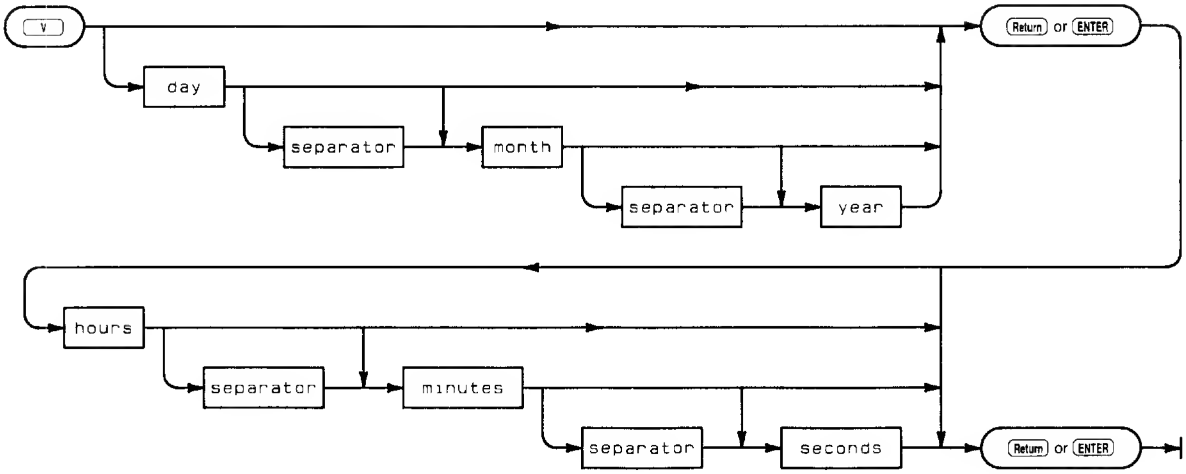
Semantics

Included in the meaning of ''program'' are user programs and operating system programs such as Editor, Filer, Compiler, etc.

Global variables are zeroed at the time a program is loaded, not each time a program is rerun. However, note that neither local variables nor dynamic variables are zeroed.

Version

The Version operation allows you to change the system time and date.

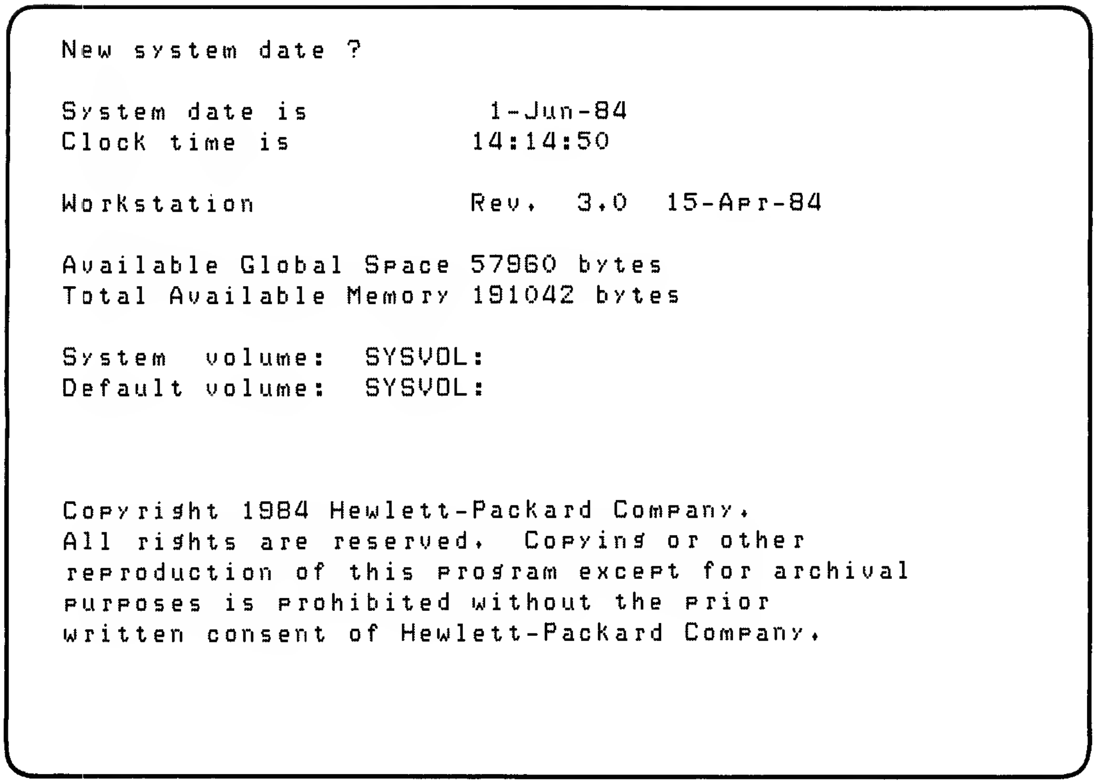


Item	Description/Default	Range Restrictions
day	integer	1 thru 31
month	three alpha characters; letter case is ignored	Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
year	integer	0 thru 99
hour	integer	0 thru 23
minutes	integer	0 thru 59
seconds	integer	0 thru 59
separator	non alphanumeric character	:, -, /, space, etc.

Semantics

In addition to prompting for the system time and date, some operating system information is displayed. The current operating system revision, available global and user memory space information is displayed. Also, default and system volume information is displayed.

The Version Prompt



```
New system date ?

System date is          1-Jun-84
Clock time is          14:14:50

Workstation            Rev.  3.0  15-Apr-84

Available Global Space 57960 bytes
Total Available Memory 191042 bytes

System  volume:  SYSVOL:
Default volume:  SYSVOL:



Copyright 1984 Hewlett-Packard Company.
All rights are reserved.  Copying or other
reproduction of this program except for archival
purposes is prohibited without the prior
written consent of Hewlett-Packard Company.
```

For more details on "Global Space", see the Compiler chapter.

Software Revision Note

This note describes the Pascal 3.01 revision. For future reference, you may want to insert it into the System History section of the Technical Reference Appendix to the *Pascal 3.0 Workstation System* manual.

Pascal 3.01

The purpose of this revision is to fix bugs in version 3.0 of the Pascal system. The 3.01 BOOT: and ASM: discs contain software which corrects the bugs. (Note that other discs have not been revised.)

Note

These revisions do *not* add any features to the system; they only fix bugs in existing features.

Documentation Changes

Since the 3.01 software does not add any features to the system, you may replace references to the 3.0 BOOT: and ASM: discs with references to the 3.01 discs.

Disposition of 3.0 BOOT: and ASM: Discs

If you have version 3.0 BOOT: and ASM: discs, replace them with the 3.01 discs. **Do not use the old discs any longer.**

List of Bugs Fixed

Here are the areas in which bugs have been fixed by the 3.01 revisions:

- Flexible disc initialization on Model 226 and 236 Computers equipped with an HP-UX Memory Management processor board and the 3.0 Boot ROM.
- Softkeys and bus errors while using the Debugger.
- Disassembly of shift and rotate instructions with the REVASM module.
- Model 237 display driver module (CRTB).
- Non-advancing characters on some foreign language keyboards.

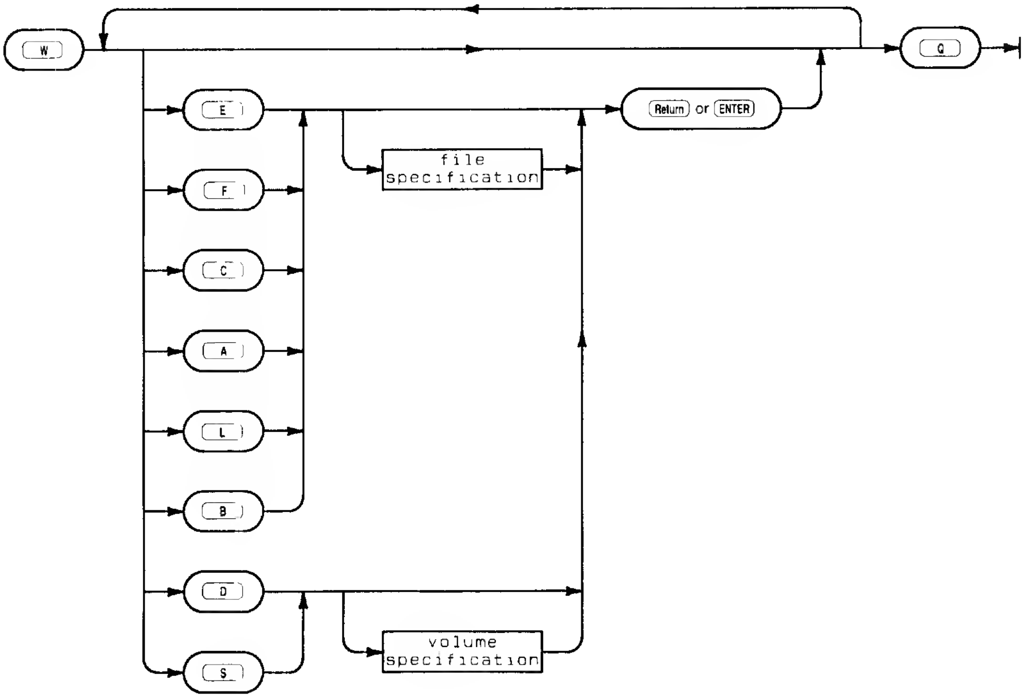
Fort Collins Systems Division • 3404 East Harmony Road • Fort Collins, Colorado 80525

98615-90080
E 1084
October 1984
Printed in U.S.A

HEWLETT
PACKARD

What

The What command displays the "system file table" and allows you to specify new file specifications for the system files.

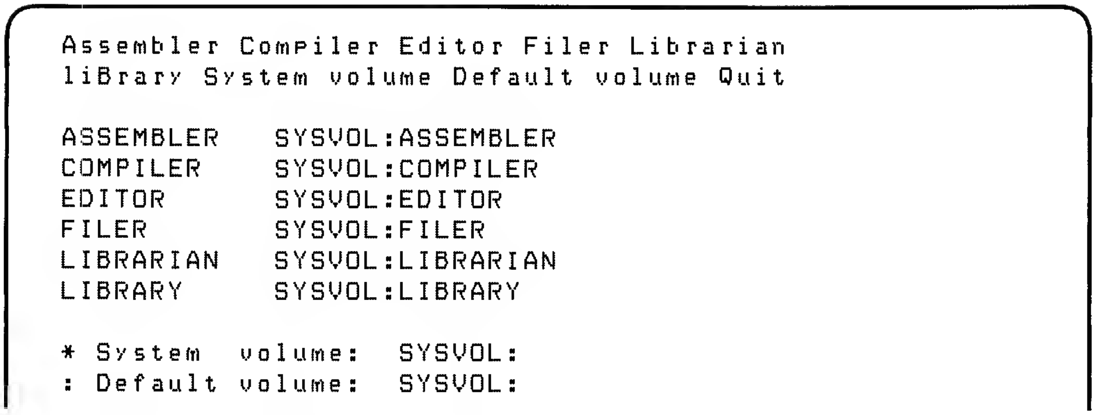


Item	Description/Default	Range Restrictions
file specification	literal	any legal file specification (see the File System chapter)
volume specification	literal	any legal volume specification (see the File System chapter)

Semantics

The system file table contains file specifications that are used by the operating system when locating system files (Assembler, Compiler, Editor, Filer, Librarian, Library, and Default and System volumes). The What command displays the system file table; a typical example is shown below.

The What Display



```
Assembler Compiler Editor Filer Librarian
liBrary System volume Default volume Quit

ASSEMBLER   SYSVOL:ASSEMBLER
COMPILER    SYSVOL:COMPILER
EDITOR      SYSVOL:EDITOR
FILER       SYSVOL:FILER
LIBRARIAN   SYSVOL:LIBRARIAN
LIBRARY     SYSVOL:LIBRARY

* System  volume:  SYSVOL:
: Default volume:  SYSVOL:
```

Typing one of the uppercase letters at the top of the menu allows you to change the corresponding file specification for that system file.

Note

When specifying a system file name that does not have a `.CODE` suffix, use a period at the end of the file name to prevent a `.CODE` suffix from being appended to the file name.

With this command, it is possible to do such things as specify a file other than LIBRARY as the System Library or your custom graphics editor as the System Editor. In the case of your custom editor, you need only press E to invoke it.

Specifying a logical unit number, such as `#3:`, as the Default volume allows *any* disc media in a unit with removable media to be the desired volume. To subsequently specify any volume in the default unit, only the file name need be specified. To accomplish this, make sure that the drive door is open, type D for a Default volume change, and then type the following:

```
#3: Enter
```

The File System

Chapter

2

Introduction

This chapter introduces you to the Pascal File System. The File System organizes and accesses information which is stored on mass storage devices. Even if you are an experienced programmer, you should read this material because it will help you understand the features of your Pascal workstation.

Your computer has built into it a substantial amount of very high speed memory called Random Access Memory, or **RAM**. This memory is called **primary storage** to distinguish it from external mass storage called **secondary storage**. Normally, data processed by the computer must first be placed in internal memory. (The term "data" is used broadly to mean any information processed by the computer, so programs are data too.) RAM has three important characteristics:

- RAM is very fast: Some data items can be stored or retrieved from RAM in less than a millionth of a second.
- RAM is volatile: Data in RAM is lost when the computer is powered off.
- It is expensive compared to alternative, slower forms of data storage, such as discs or magnetic tape.

Information not immediately needed by the computer is kept in secondary storage. Some important characteristics of magnetic discs are:

- Data access is slow compared to RAM, often as much as ten thousand times slower.
- The data is relatively permanent, that is, it is available until erased.
- Magnetic storage is inexpensive compared to RAM.
- Magnetic media are often removable and replaceable, providing an almost unlimited amount of long-term storage.

How Magnetic Discs Work

Discs come in two types, "flexible" and "hard". Flexible discs are also known as "floppy discs" since they are light, thin and can be bent slightly. Hard discs are sometimes called "fixed", since the disc is not removable from most hard disc drives.

Both types of discs work in essentially the same way. The disc is a platter similar to a phonograph record made of plastic or metal. The disc is coated with a smooth layer of microscopic magnetizable particles similar to that used in tape recorders. When the disc is in a disc drive, it spins very fast. As it spins, a magnetic sensor similar to the recording/pickup head in a tape recorder is held over the disc's surface. The disc drive has a mechanism used to move this head over various parts of the disc's surface.

The recording groove in a phonograph record is a continuous spiral from the outer edge to the middle. By contrast, magnetic discs are organized into a sequence of concentric but unconnected circular tracks. The computer must tell the disc drive where to place the head over a particular track in order to read or write data. The tracks themselves are logically broken up into blocks of data called sectors. Discs are often referred to as "blocked devices" because of this structure.

The smallest amount of data that can be read from or written to a disc is a single sector. HP LIF discs have sectors with length capable of holding 256 bytes (characters) of information. HP WS1.0 discs have sectors with length capable of holding 512 bytes (characters) of information. The computer may read or write several sectors in immediate succession. Since the disc is spinning, once the recording head is positioned over the correct track the computer must still wait until the desired sector rotates into position under the head. By processing one sector after another as fast as the disc is rotating, the time delay caused by waiting for the sector to get into the correct position can be effectively eliminated.

For various reasons, the computer may not be ready for the next sector as it spins into position. By staggering the sectors on the disc it is possible to insure that the next logical sector rotates into place just when the computer is ready for it. This staggering technique is called **interleaving**, and it can greatly improve your system's performance. Using the wrong interleave factor can likewise drastically reduce your system's performance.

For example, imagine a track that has 16 sectors of data numbered 0 through 15. If the disc has an interleave factor of one, the sectors are simply in order:

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
```

After reading sector zero, the computer must immediately be ready for sector one. If the computer isn't ready for sector one, it will be missed and sectors two through fifteen and zero will pass under the head before sector one is again accessible. Thus a single sector would be read on each disc rotation. This is not very efficient.

Now suppose the computer's busy period after reading a record is just a little less than the time that elapses while the next sector passes under the head. By placing sectors out of order on the disc as follows:

```
0  8  1  9  2  10  3  11  4  12  5  13  6  14  7  15
```

the computer can access sector zero, miss sector eight, access sector one, miss nine, and so forth. It is not necessary to wait for an entire disc rotation between successive sectors. The numbering scheme shown is said to have interleave two, since looking at every other sector accesses them in logical sequence.

The interleave factor for flexible discs is established by a process called initializing (some manufacturers use the term "formatting"), which must be done before the disc is used. Initializing is done by a utility program called MEDIAINIT supplied with your Pascal system. MEDIAINIT knows the appropriate interleave factor to use with various models of disc drives.

The default interleave for the disc you are initializing is shown in one of MEDIAINIT's prompts. This default is the best interleave factor for that particular device. For example, an HP 8290X defaults to an interleave factor of 3. For hard discs, the interleave factor is established at the factory; initialization of hard discs serves mainly to find bad tracks and force the use of spare tracks if necessary, not to change the disc's interleave.

Pascal Volumes

Let's take an exploratory trip, using the computer itself to investigate the file system. You should already know how to load the Pascal Language system, and be aware of its various subsystems, such as the Editor and Filer.

To begin our journey, begin at the Main Command and load the Filer subsystem by pressing:

F

The Filer is located on the Pascal disc labelled **ACCESS:**. When the Filer's prompt line appears, execute the Volumes command by pressing:

V

The various disc drives connected to your computer will be accessed and then you will see a display similar to the following:

```
Volumes on-line:
 1     CONSOLE:         This is your workstation's CRT display.
 2     SYSTERM:         This is your workstations keyboard.
 3   # BOOT:            BOOT:  disc is in right-hand drive.
 4   # WRKING:          Initialized disc with volume name WRKING:
 6     PRINTER:         Printer is connected to built-in HP-IB.
11   * SYSVOL:          The * indicates the system volume.
12   # MYVOL:           Volumes 11 through 14 are examples of a
13   # MASTER:          possible HP 9134 disc drive configuration.
14   # V14:
50   # RAM:             Ram volume made with Memory Volume command.
Prefix is - MYVOL:      MYVOL:  is current prefix volume.
```

Precisely what will be displayed depends on what peripherals are connected to your computer and what discs are currently installed in the disc peripherals. Note how your display appears. You may want to change some of the discs in your systems disc drives or turn off a peripheral and see how that changes the display.

Volumes

The word "volume" was chosen by analogy to a book. Volume denotes a logical entity in which a substantial amount of information can be stored. For instance, a flexible disc is a volume. Volumes have names by which we may refer to them. The display above shows what volumes are currently accessible to the file system on my system.

Notice that each volume name is followed by a colon. This convention is used throughout the Pascal system. The colon is a delimiter or punctuation mark which separates the volume name from further information used to designate data within the volume.

A single large disc may contain more than one volume, as a shelf can hold more than one book. Flexible discs contain a single volume. For flexible discs we may use the volume name as the disc drive's name. By that I mean, if we refer to the volume `BOOT:` by name, the computer will find it in whichever drive it is located.

Note

Because the file system works with named volumes, it is very important not to have more than one volume of a given name on-line at one time. The file system may destroy data by using one volume when you meant the other.

You can see that some of the volume names don't correspond to any disc device at all. Actually the file system has a name for each input/output device. SYSTERM: is the name of the keyboard volume, CONSOLE: is the name of the CRT, and PRINTER: is the name of the system hard copy device. We will have more to say about these non-disc volumes later.

Logical Units

The numbers in the column to the left of the Volume names displayed above are called "logical unit numbers" or simply "units". The volume name denotes a particular disc, while the unit number denotes a particular location for a volume. In the case of flexible discs, the unit number corresponds to a physical disc drive. In the case of a large fixed disc which is divided into several logical volumes, each logical unit represents a portion of the disc surface which is treated as if it were a separate physical disc drive.

To refer to a unit instead of a volume, use a # followed by the unit number. For instance `#3:` and `BOOT:` both refer to the same volume as long as BOOT: is installed in the right-hand disc drive of a Model 226 or Model 236 computer.

Drive Numbers vs. Unit Numbers

Since a single machine can contain two or more drives, you need to be able to distinguish between them. If you read the machine's manual, you will find that the drives are differentiated by *drive number*. For instance, the right-hand floppy drive in a 9836 is drive number 0, while the left-hand drive is drive number 1. The File System distinguishes between them by assigning each a unique *logical unit number*. In the case of the 9836, these drives are normally assigned unit numbers 3 and 4, respectively. With external dual floppy drives, drive 0 is usually the left-hand drive, while drive 1 is the right-hand drive. And with hard disc drives, there can be several drive numbers. Don't be alarmed, however, because the system takes care of the correspondence between drive numbers and unit numbers for you. In addition, this manual refers almost solely to logical unit numbers, not drive numbers. Drive numbers were mentioned so that you would realize that they are not the same as unit numbers.

Blocked and Unblocked Units

Some of the units are displayed with # or * between the unit number and the volume name. These are blocked units. Blocked units are memory devices that are divided into sectors (blocks) and have directories describing their contents.

We aren't yet ready to talk about the data stored in a volume, but you probably won't be surprised to learn that it is organized into groups called "files", which are like chapters in a book. The directory of a blocked volume is essentially a table of contents.

The other units are unblocked or "byte stream" devices (such as the printer, keyboard, and CRT). Unblocked devices process information one character at a time and do not have directories.

The System Volume and Default Volume

Although your workstation can deal with many volumes (up to 50 on-line at once), there are two volumes which are referred to so frequently that special abbreviations have been provided to name them. They are the *system volume* and the *default volume.*

The System Volume

The system volume is used by the Operating System to store its own private files and records. Since the Operating System is always overseeing your computer's operation, the system volume needs to be accessible practically all of the time. The abbreviated name for the system volume is * (asterisk), which appears next to the system volume in the Volumes command's display. The asterisk need not be followed by a colon, since it is distinctive. Thus for the Volume display shown previously, these notations all denote the system volume:

```
*
*:
SYSVOL:
#11:
```

Here are some of the ways the operating system uses the system volume:

- When the Operating System is loaded and begins to function, it looks on the system volume for subsystem programs such as the Editor, Filer and Compiler. If these subsystem programs are on other volumes, the Operating System will still find them.
- When the Operating System first begins to function, it looks on the system volume to find the system date. The system date is put on all files as they are created to help in maintaining file organization. If you change the system date, the new date gets written on the system volume.
- During processing of a stream file, data may be temporarily stored on the system volume. A stream file is a pre-recorded sequence of keystrokes which are treated as if they came directly from the keyboard.
- If you create an anonymous file (see the Programming With Files section), it will be stored on the system volume. An anonymous file is a file created by a program, used by the program, and then destroyed when the program ends. While the program is in existence, the anonymous file is for all purposes a real file.
- If you use a work file during development of a program, it will be stored on the system volume.
- If you use an AUTOSTART or AUTOKEYS file, it must be stored on the system volume.

The Default (Prefix) Volume

The other special volume is the default volume. This volume is sometime called the prefix volume. In many applications it is most convenient to have the frequently needed files together in a single volume. If these files are being accessed frequently, it is tedious to constantly type the volume name or unit number. You can instead tell the system that when no volume name is specified, the one to use is the default volume. You specify the default volume by using the Filer's Prefix command.

The preceding Volumes display indicates "Prefix is - MYVOL:". This means that MYVOL: is the default volume. The default volume can be specified in two ways. If a colon separator appears with no volume name before it, the default volume is assumed. If a file name is given with no volume name before it, the default volume is assumed.

Use the Filer's Prefix command to set the default volume name:

```
P
Prefix to what directory?
SYSVOL: Return or ENTER
Prefix is SYSVOL:
```

The default volume and the system volume can be the same volume. In fact, except for single drive configurations, the default condition you received from the factory has SYSVOL: as both the default and system volume.

If a unit specified by unit number (e.g. #3:) in a Prefix command does not contain a disc when the Prefix command is executed, that unit becomes the default volume. That means that the current disc in that drive, no matter which disc is the current disc, is the default disc as long as it is in the drive.

You can also set the default volume name using the **What command** of the main Commmand level. The What command is more powerful than the Prefix command because What allows you to specify a new system volume, as well as the name and location of each of the system files (Filer, Editor, Library etc). For further information on the What command, see Chapter 1 of this manual.

Files

Information within a blocked volume is further organized into **files**. A file is a collection of related information, having a name by which it is identified during the file operations. Since a volume usually contains many files, within the volume there is also a directory, or "table of contents," telling the name of each file, how big it is, what sectors it occupies on the disc, and (roughly) what sort of data it contains.

Files are created by computer programs – either system programs (such as the Editor, Filer and Compiler,) or user application programs.

For example, when you save a Pascal program written with the Editor, the program is saved with the specified file name in either the current default volume or the specified volume. When that same program is compiled, the object code is stored in another file. When the object code program is executed, it may create more files.

You can use the Filer to list the files in a volume. For instance, to see what is in the default volume of our example system, type:

```
L
```

to invoke the Filer's List directory command. The Filer responds with:

```
List what directory?
```

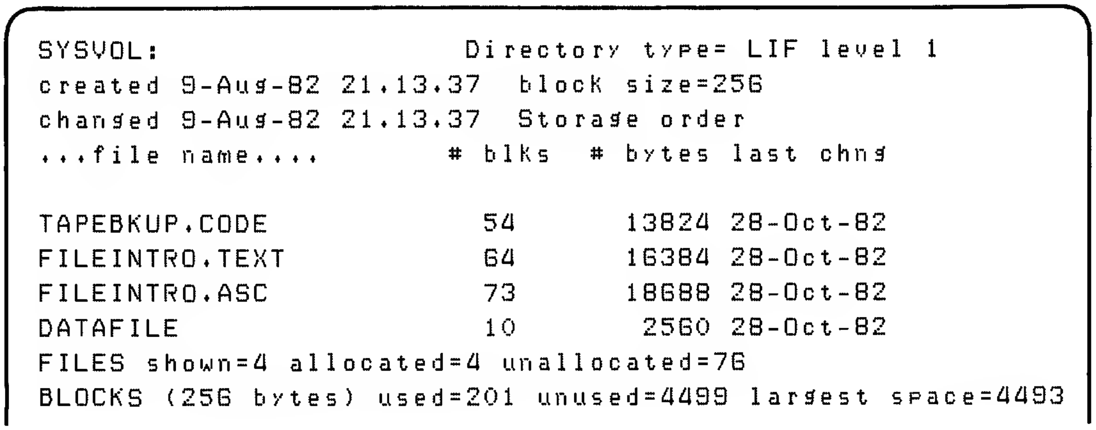
When a volume is **not** specified, the default volume is assumed. To specify the default directory, type in:

: Return or ENTER

Assuming the configuration shown on previous page, you could have done the same job by typing:

SYSVOL: Return or ENTER

The listing of the default volume's directory is shown below.



```
SYSVOL:                      Directory type= LIF level 1
created 9-Aug-82 21.13.37  block size=256
changed 9-Aug-82 21.13.37  Storage order
...file name....          # blks   # bytes last chng

TAPEBKUP.CODE               54       13824 28-Oct-82
FILEINTRO.TEXT              64       16384 28-Oct-82
FILEINTRO.ASC               73       18688 28-Oct-82
DATAFILE                    10        2560 28-Oct-82
FILES shown=4 allocated=4 unallocated=76
BLOCKS (256 bytes) used=201 unused=4499 largest space=4493
```

File Naming Conventions

The definition of HP Pascal tries to minimize the work of moving Pascal programs from one operating system to another by requiring the use of string values to specify the names of files and certain other information such as passwords and access rights.

In Pascal 2.0 and later versions, the allowable syntax of a file name depends on the type of directory in which the file resides. The underlying file support is structured to allow programs to work properly regardless of the directory organization(s) being used, but the syntax of file names is defined by the type of directory on the volume.

File Specifications and File Names

There is a difference between a file specification and a file name. A file name is a character string which is the external identifier by which a file is designated in a disc directory. A file specification is a character string which consists of the file name and several other optional items: volume_id, directory_path, passwords, and size_specifier. Not all of these items are allowed by every Directory Access Method or under all circumstances; for instance, directory paths and passwords are only used with the Shared Resource Management System's hierarchical directory organization.

Syntax of a File Specification

The syntax of a legal file specification is given by the following diagram:

```
file_specification ::= [volume_id] [directory_path] file_name ["["size_spec"]"]

                   ::= volume_id
```

In this notation, items between square brackets [and] are optional; quoted items appear literally. The definition just given means that a file_spec (file specification) may appear in one of two forms. The first form consists of an optional volume_id followed by a colon, then an optional directory_path, then a file_name which is not optional, then an optional size_spec. The second form consists just of a volume_id.

Examples of the first form are as follows:

```
File_x
A49ZB[10]
#4:LIBRARY.
BOOT:SYSTEM_P
#45:SYSTEM21/FILER
*EDITOR.
```

Examples of the second form are as follows:

```
BOOT:
#3:
*
:
#45:SYSTEM21/TOOLS
#45:
```

Syntax of a Volume Identifier

The volume_id selects one of up to 50 logical units known to the file system. If no volume_id is present, the volume used is the "default volume" selected by the Filer's Prefix command. Otherwise, the volume is specified in one of two ways:

```
volume_id ::= "#" integer [ password ] ":"

          ::= name [ password ] ":"
```

In the first case, the integer is a two-digit number from one to fifty; for example, #23: is a volume_id. In the second case, the name is a sequence of characters. The length of the name and allowable characters depend on the particular directory organization used by the logical unit. For mass storage devices, the volume name is actually stored on the disc itself so it can be identified whenever it is inserted into a drive. For devices which have no directory, such as printers, the volume name is an arbitrary one supplied by the TABLE configuration program at boot-up time.

Example volume_ids of the second form are MYSYS: and PRINTER: Volume_ids may be 6 characters long in LIF directories, 7 characters long in Workstation 1.0 (UCSD-compatible) directories, and 16 characters long in SRM directories. LIF and SRM allow lowercase letters, while WS1.0 and unblocked devices ignore letter case. WS1.0 converts all characters to uppercase automatically.

In the case of a logical unit connected to a Shared Resource Management System, the volume_id takes a special meaning. The notation #5: refers to the current working directory of unit number five; the notation #5:/ refers to the root directory of the SRM with which unit number five is associated. The current working directory for any SRM volume is selected by the Filer's Prefix or Unit commands, or the What command of the Main Command level.

On the other hand, if the logical unit does not have a hierarchical directory, then the two volume_id notations (e.g., #11: and SYSVOL:) have the same meaning. This is the case for all local mass storage devices.

Syntax of a Directory Path (SRM)

Directory paths are only allowed when specifying files on SRM logical units. The syntax for a directory_path is:

```
directory_path ::= [ "/" ] { directory_name [password] "/" }

password ::= "<" word ">"

directory_name ::= file_name
               ::= "."
               ::= ".."
```

The use of curly braces "{" and "}" indicates that the information between them may occur zero or more times. As you can see, there are two special directory names allowed with the SRM. The name "." (a single period) refers to the current directory somewhere along a path to a file of an SRM logical unit. The name ".." refers to the parent of the current directory. Other file names occuring in a directory path are directories along the path to the one which contains the file being specified.

Passwords are sequences of up to 16 characters, which govern the access rights to a file or directory. They are given to a file either at creation time or by use of the Filer's Access command.

Note that a directory path doesn't appear by itself; it appears as part of a file specification, with the file name after the directory path. Examples of directory paths are:

/.<PASS1>/	Denotes root, using password "PASS1".
/USERS/ROGER/	Denotes directory ROGER in USERS, which is in root directory.
HERE/THERE/	Denotes directory THERE, found in HERE.
../THERE<PASS2>/	Directory THERE, found in the parent of the current working directory.

A directory path together with a volume_id might appear as follows:

```
#5:/WORKSTATIONS/SYSTEM13/
```

Occasionally there is need for a volume password, which is a case not covered by the above syntax. You may use either of the following forms:

```
#5<volpassword>:/dirname1/dirname2/filename
#5:<volpassword>/dirname1/dirname2/filename
```

That is, the volume password may either immediately precede or follow the colon separator.

Syntax of File Names

To the Pascal Workstation System, a file name is just a sequence of characters. The Directory Access Methods allow all printable characters. However, the following characters have significance either in Filer commands or in the overall specification of files under various Directory Access Methods (such as directory paths in hierarchical directories), and therefore should be avoided in file names:

- sharp '#'
- asterisk '*'
- comma ','
- colon ':'
- equals '='
- question mark '?'
- left bracket '['
- right bracket ']'
- dollar sign '$'
- less than '<'
- greater than '>'

Control characters (ASCII ordinal value less than 32) and blanks are removed by the File System before the name is ever presented to any Directory Access Method.

File Types Derived from File Names

The type of a file is determined when it is created, and is derived from a suffix (the last characters of the file name). Once the file type is determined, a type code is recorded in the directory, and changing the file name won't change its type.

Suffix	File type
.ASC	LIF ASCII text file
.TEXT	WS 1.0 / UCSD compatible text file
.CODE	Pascal 2.0 object code
.BAD	File covering bad area of disc
.SYSTM	Boot image file
No Suffix	"Data" file

File Names (LIF)

The LIF Directory Access Method (DAM) generally allows any ASCII character to be used in a file name. This is contrary to the HP LIF Standard, which states that file names must be composed only of upper-case letters, digits, and the underscore '_' character. Note that upper and lower case letters are distinct. File names stored in LIF directories are always exactly 10 characters.

The LIF DAM recognizes only upper case suffixes.

The 10-character file name length would be a very severe restriction when four or five characters are required for a suffix. To ease this problem, the LIF DAM performs a transformation on the file name which compresses the suffix if one is present. The transformation occurs automatically when a LIF directory entry is made, and it is reversed automatically before the file name is ever presented to any program or to the user.

This process is usually completely transparent to the Pascal user, although its effects may be seen when a LIF directory is examined from the BASIC language system. It sounds complicated and dangerous, but in practice it is very smooth. Most people would never notice it if they weren't told.

Here is how the LIF DAM changes a name before putting it into the directory.

1. Look for a standard suffix (for example, ".ASC").
   a. If a suffix is found, the suffix characters are removed from the name, leaving a trailing period. If this name is longer than 10 characters, including the period, then an error is reported.
   b. If no suffix is found, and the file name contains less than 10 characters, the file is assumed to be a Data file and the name is put into the directory unchanged. If no suffix is found, but the file name is exactly 10 characters in length and the last character is an A, B, C, S, or T, then an error is reported.

2. If the file is not a Data file and no error has been reported, the dot is replaced by the first letter of the suffix; for instance, the .ASC suffix is replaced by A. If the name is now less than 10 characters long, it is extended to a length of 10 characters by appending underscore characters (_) to the name.

Using this algorithm, we would have the following examples:

File name	Translated name
'A.ASC'	'AA________'
'charlie'	'charlie '
'123456789.TEXT'	'123456789T'
'GollyGeeeT'	rejected because it would be confused with transformation of 'GollyGeee.TEXT'

The reverse transformation is fairly obvious:

1. If the 10th character is a blank, do nothing; otherwise,
2. Remove all trailing underscores.
3. Compare the last non-underscore to the first letter of each valid suffix. If a match is found, remove that letter from the file name and append a dot '.' followed by the full suffix.
4. If no suffix match is found, use the original file name.

File Names (Workstation 1.0 Directory)

The Workstation 1.0 (UCSD compatible) DAM allows file names of up to 15 characters including the suffix. Any lower-case letters are transformed to upper-case, so that 'a.text' and 'A.TEXT' denote the same file.

File Names (SRM System)

The SRM itself allows almost any file name. The Pascal system removes blanks and control characters from file names.

However, the Pascal SRM Directory Access Method takes the "<" character to denote the beginning of a password. All characters up to the next ">" character are part of the password, so that <<<<<<<<> is a (poorly chosen) password. Passwords may be up to 16 characters long.

File Size Specification

The last, optional part of a file specification is the file size specifier. If present, its syntax is

```
size_spec ::= "[" integer "]"
          ::= "[*]"
```

This specification only takes effect if a new file is being created with REWRITE, OPEN,APPEND or APPEND with OPEN . If the file already exists, the file system tries to make it at least the size specified. The size is ignored for RESET.

In the first form, the integer gives the number of 512-byte blocks to be allocated to the file. For instance [100] would cause allocation of 51 200 bytes.

The second form [*] specifies that the file is to be allocated either (half of the largest free space) or (the second largest free space), whichever is larger.

If no size specifier is present when space for a new file is being allocated, the largest free area is assigned to the file.

For files stored in the SRM, the first extent allocated to the file will be contiguous and of the size specified if possible.

Several Directory Organizations Allowed

HP LIF (Logical Interchange Format) is the default directory format used by your Pascal Language System. There are many (mutually incompatible) ways to organize files and directories on a disc. LIF is an HP standard disc organization used to transport files between computers made by Hewlett Packard Company. The HP Series 200 BASIC Language System also supports the LIF directory structure on your Series 200 computers.

In addition, your Pascal workstation understands two other disc directory organizations. The WS1.0 format was the primary disc directory format used by the Pascal 1.0 Language System. You Pascal workstation also supports the hierarchical directory structure used by the Shared Resource Management System. The SRM and hierarchical directories are discussed as a separate topic later in this section. The WS1.0 format is compatible with the widely used UCSD* Pascal system.

File Name Suffixes and File Types

Our example default volume contains four files: TAPEBKUP.CODE, FILEINTRO.TEXT, FILEINTRO.ASC and DATAFILE. The first three file names have a suffix. This suffix is part of the file name, so FILEINTRO.ASC and FILEINTRO.TEXT are different files. The suffix was appended to show the file type when the file was created. The file type, once determined, is stored in the directory along with the file name. That means the file type would not be changed if you later changed the file name and removed or changed the suffix. You can see the file type of each file by listing the directory using the Filer's Extended Directory List command.

The suffixes recognized by the Pascal 2.0 (and later) File System are shown below:

.ASC — Information stored in an .ASC file is stored as individual strings. Each string has a two-byte string length header. Text files are produced by the Editor.

.TEXT — .TEXT files follow the WS1.0 or UCSD Pascal format. They Have a 1024-byte header containing environment infomation. This header is followed by compacted text in 1024-byte pages. Each line of information begins with ASCII characer 16. This is followed by an integer calculated to be 32 plus the number of initial blank characters in the line. Each line is then terminated by ASCII character 13. If a line will not fit in a 1024-character page, the whole line is moved to the beginning of the next page and the remainder of the previous page is filled with blanks.

.CODE — A .CODE file is the object code produced by the Compiler, Assembler or Librarian. This file is also called a library.

.SYSTM — A .SYSTM file is a special file recognized by the Boot ROM as a file containing an operating system.

.BAD — A .BAD file is used to cover failed disc sectors. Use the Filer's Make command to make a file of type .BAD over the defective sector of the disc media. This type of file should only be used as an emergency measure. The defective disc should be replaced.

A file whose name at the time of creation does not end in one of these suffixes is said to be of type DATA.

* UCSD Pascal'' is a trademark of the Regents of the University of California.

The Pascal system, in many circumstances, automatically appends the appropriate suffix to a file's name. For instance, when loading a file into the Editor, just type the file name without the suffix. The Editor knows that in normal circumstances you will want to edit a .TEXT file and will automatically add the suffix. Of course, if you wish to type the suffix you may. If you want the Editor to load another type of file, then the correct suffix must be specified. The period stops the Editor from adding the suffix .TEXT. If you try to specify a file type that the subsystem can't work with, such as a .CODE file in the Editor, you will get an error message reading

```
Undefined operation
for this file/unit.
```

Automatic suffixing is very convenient. For instance, you might write a program with the Editor and call the output file WORK. The Editor automatically appends .TEXT. When you use the Compiler to compile WORK, the Compiler automatically appends .TEXT to the source file name, and .CODE to the output file name. Although there are two files, you only need to remember one name. To execute WORK.CODE, you need only press R or RUN; alternately, you could type the following (from the main command level):

Select (EXECUTE)
WORK Return or ENTER

Suppressing the Suffix

On the other hand, you may wish a file name which has no suffix. You can suppress the automatic appending of a suffix by typing a period as the last character in the file name. For instance, to create a data file with the name AFILE, just tell the Editor to save your file as `AFILE.`. The period aborts the suffix and makes the type DATA. Likewise the Librarian and Compiler will automatically append .CODE to file names unless you tell them not to with the period.

System programs like the Editor don't have the .CODE suffix. This protects them against accidental destruction by a wildcard purge operation on all .CODE files. If you wish to permanently load a system program into memory with the Permanent command, you must append a dot to the file name. To load the Editor, type:

P
EDITOR . . Return or ENTER

Without the period, the system would try to load EDITOR.CODE, which is not what you want.

Translating Files from One Type to Another

Sometimes you may want to translate the contents of a file from one file type to another. For instance, you may have a file of type .TEXT, created by the Editor, and wish to read it with BASIC 2.0. BASIC 2.0 understands the LIF ASCII (.ASC) format, but not the .TEXT format. You can use the Filer's Translate command in this situation. A typical dialogue would be:

```
T
Translate what file?
EXAMPLE.TEXT  Return or ENTER
Translate to what?
EXAMPLE.ASC Return or ENTER
```

For Translate to make sense, the source file must contain data that is textual in nature; attempting to translate a .CODE file, for example, would not make sense.

Another situation where translation is required is to move a file from a disc volume to the printer. The file may be a .TEXT, .ASC or DATA file . The way such files are stored on the disc is not compatible with unblocked devices (such as the printer), so you must use the Translate command. Just type in:

```
T
Translate what file?
WORK.TEXT Return or ENTER
Translate to what?
PRINTER: Return or ENTER
```

This example illustrates several points. First, in the Filer environment you must always specify the complete file name including the suffix. Second, to send a file to a device like the printer which has no directory, there is no point in specifying a file name. Just use the volume name. Had you specified a file name after PRINTER: the Filer would have given you an error message.

Wildcards

In the Filer environment you can specify a particular file or set of files by giving a pattern which identifies the files you want. These patterns include special characters called **wildcards**.

For example, we can use the wildcard = (equal sign) to list a subset of the file using the Filers List Directory command. From the Filer subsystem, press:

```
L
```

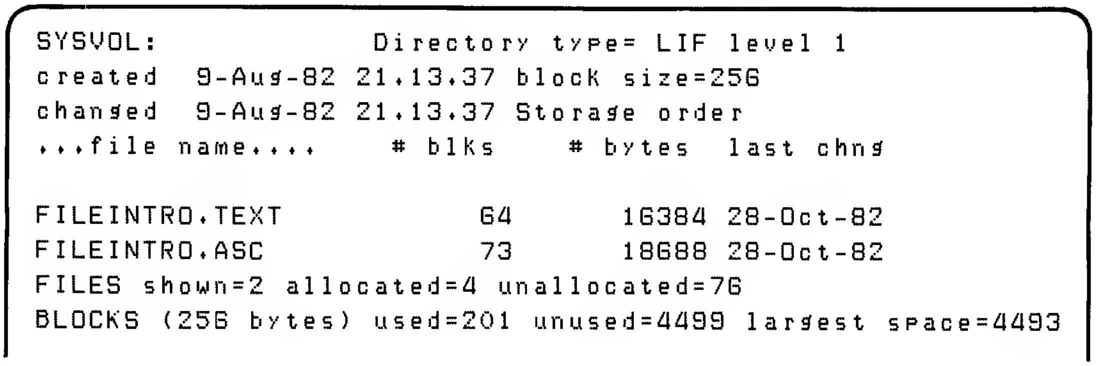
The computer responds with:

```
List what directory?
```

Respond with:

```
FILE= Return or ENTER
```

`FILE=` uses the equal sign wild card to specify all files whose names begin with FILE and end with any sequence of characters. Using our example system, this command sequence would produce:



```
SYSVOL:               Directory type= LIF level 1
created  9-Aug-82 21.13.37 block size=256
changed  9-Aug-82 21.13.37 Storage order
...file name....     # blks     # bytes  last chng

FILEINTRO.TEXT            64       16384 28-Oct-82
FILEINTRO.ASC             73       18688 28-Oct-82
FILES shown=2 allocated=4 unallocated=76
BLOCKS (256 bytes) used=201 unused=4499 largest space=4493
```

Notice what happened here. The Filer recognized that the response to the prompt, "List what directory?", specified not just a volume name but a set of files within that volume.

More than one wild card may appear in a single file specification given to the Filer, allowing you to easily describe some rather complex operations. For instance, you can copy all the files on unit #13 whose names contain the characters INT to the system volume by means of this command sequence:

```
(F)
Filecopy what file?
#13:=INT= (Return) or (ENTER)
Filecopy to what?
*$ (Return) or (ENTER)
```

This example uses the destination wildcard "$", which means "use the same name as the source file had". The command locates each file on unit #13 whose name matches the pattern, and writes a new copy with the same file name on the system volume. Remember that * is shorthand for the name of the system volume.

You can use "?" as a wild card instead of "=". Question mark works like equals, except that for each file whose name matches the specification, the Filer will ask if you want to perform the operation. For example, to have the Filer change each file name on the default volume beginning with FILE into a file name beginning with WORK, type in:

```
(C)
Change what file?
FILE? (Return) or (ENTER)
Change to what?
WORK= (Return) or (ENTER)
```

This would for example turn FILE_ONE.TEXT into WORK_ONE.TEXT. Each time the specification is met, the Filer will present what it has found and ask if the process should be completed for the entry. Answer with Y for yes or N for no each time you're asked.

File Names to Avoid

The file system won't prevent you from creating file names containing wildcard characters, but you'll be sorry if you do. The Filer will think such file names are wildcard specifications instead of simple file names. For instance if you created a file called =.TEXT, then used the Filer sequence:

```
(R)
Remove what file?
=.TEXT (Return) or (ENTER)
```

the Filer would remove every file whose name ends in .TEXT in the default volume!

Should you ever accidentally create a file with a wildcard in its name in volume VOLNAM, you can get rid of it this way:

```
(R)
Remove what file?
VOLNAM:? (Return) or (ENTER)
```

This will cause the Filer to offer to remove each file in the directory VOLNAM:. You can then remove the problem file, and retain the other files.

Allowable File Names

What file names are allowable depends on the type of directory used on the volume in which the file resides. In other words, the directory organization makes the file name rules. The exact rules for file names are given in the section Programming with files in this chapter. Here is a summary of the rules.

It is wise to choose names consisting of alphabetic letters and digits; if you want a punctuation mark within a file name, a hyphen, an underscore, or period is acceptable. Blanks are removed from file names.

In LIF directories and SRM directories, upper and lower case letters are distinct; "CHARLIE" is not the same file as "Charlie". In WS1.0 directories lower-case letters in a file name will automatically be converted to upper-case. This exception makes it easier to use wildcards to move files from one type of directory to another. Only upper case suffixes are allowed in LIF directories. Lower case suffixes in LIF directories will cause an error.

Don't use the following characters:

"$", "?", "="	Filer wildcard characters.
"*", ":", "#"	Used in specifying volumes.
"/", "<", ">"	Have special meaning with the Shared Resource Manager.
"[", "]"	Used to specify the size of a file when it is created.
control characters	Control characters are automatically removed from file names.
" "	Blanks are removed from file names.

File Name Length

In LIF directories, file names (without suffix) are limited to 9 characters. If the last character in the file name is **not** an A, B, C, S, or T, then 10 characters can be used. If a suffix is present, up to 9 characters may precede the dot and suffix.

In WS1.0 directores, file names may be up to 15 characters **including** the suffix.

In SRM directories, file names may be up to 16 characters **including** the suffix.

No Room on Volume

Obviously there is a limited amount of space in a disc volume. When there is no room on a volume to create a new file, the system will report an I/O error.

You may be able to solve this problem by using the Filer's Krunch command. This command consolidates all of the volumes free space by moving all of the files on a volume to the front of the volume

Both the LIF and WS1.0 directory organizations are designed for "contiguous file space allocation". This means that when space is reserved for a file, the disc sectors set aside have sequential numbers. For instance a file requiring 3 sectors might get sectors 26, 27 and 28; or 31, 32 and 33. Files would **not** be allocated sectors 13, 56 and 2, because those sectors are not logically adjacent. To go back to the analogy with file folders in a drawer, if you had a file too big for one folder you might put it in two or three folders; but you'd want store them next to each other, not in random places in the drawer.

When a file is purged, all of its sectors are again available for use by another file. As files are created and purged, the disc space usage will develop "holes" of free space between valid files. This is called "fragmentation". It's possible for a considerable amount of free space to exist in the volume, yet be unuseable because it is in pieces too small to use. Since files tend to be small compared to the total space on a volume, this problem usually occurs when the volume has relatively little free space left.

To see how fragmented your volume is, use the Filer's Extended Directory List command. This command lists both the files and the empty space on the volume.

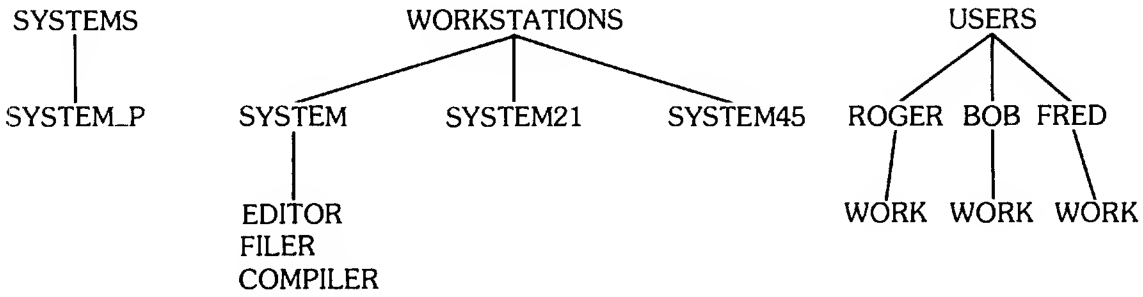
The Shared Resource Management System

The concepts presented so far have all been applied to local mass storage devices. The same concepts extend naturally to deal with shared mass storage.

The Shared Resource Management System (SRM) allows several workstations (computers) to be connected into a network that allows sharing of files and resources. This network is controlled by a system controller. Since files can now be shared between several users, a new directory structure is needed. Setting up the SRM system is not described in this manual; see the SRM documentation for that information. Configuring your workstation to access an SRM system is described in the Special Configurations chapter.

Hierarchical directories

The Shared Resource Management System uses a hierarchical directory structure to organize its files. This directory structure is a multi-way tree data structure. That is, the first, or top directory in the structure is called the **root directory**. Subordinate to the root directory are other directories which, in turn, may have further subordinate directories. Each directory may contain files or other directories. When a directory contains only files, it is called a **leaf directory**. All files can be called **leaf files**. The drawing below shows a hierarchical directory structure.



The directory SYSTEMS is a special directory used by the BOOT ROM, version 3.0 or newer, to automatically load operating or language systems.

The directory USERS has three subordinate directories: ROGER, BOB, and FRED. Each subordinate directory has a single file called WORK. Each file and directory is uniquely specified by the list of directories from the root to the file. That means several files of the same file name can exist without confusion if they are in different locations in the directory structure.

To save space, the Filer's Duplicate Link command can be used to link a file into a directory other than its original location. This allows you to have access to a file, such as the Compiler or Editor, without making an extra, unnecessary copy. See the Filer Chapter of this manual for more information.

Once a duplicate link has been set up, if the directory is purged, what happens to the link? Only the purged directory loses access to the file. All other directories with links to the file can still find it. The disc space allocated to the file is only reclaimed when no directories have links to it.

Notation

Hierarchical directories are a simple concept, but we need some specialized words and notation to talk about them.

The directory at the top of the hierarchy is called the "root" directory. If we want to refer to a file or directory which is immediately under the root, for instance WORKSTATIONS in the illustration above, we would write

/WORKSTATIONS

This is read as "slash WORKSTATIONS" or "stroke WORKSTATIONS". The / indicates the root directory.

To go further down the hierarchy, for instance to SYSTEMS under WORKSTATIONS, write

/WORKSTATIONS/SYSTEM

and for another level yet

/WORKSTATIONS/SYSTEM/COMPILER

As you can see, to specify a file, the list of directories from either the root directory or the current working directory to the target file must be specified. The list is delimited with a /.

Such a sequence of strokes and filenames is called a directory path, since it indicates the path one must follow down the hierarchy to get to a particular file.

SRM Units and Volumes

A workstation connected to an SRM normally has units #5: and #45: set up for SRM access. The use of two units is in keeping with the idea that there are usually two special volumes (the system volume and the default volume) through which most file accesses occur.

If the workstation is booted from SRM, unit #45: will automatically be configured to be the system volume and unit number #5: will be available for use as the default volume. If there is local mass storage, the system volume can be any volume you desire. To set these volumes, use the What command from the Main Command Prompt.

Here is how the Filer's Volumes display might look right after booting up a workstation connected to the SRM and having no local mass storage:

```
Volumes on-line:
   1    CONSOLE:
   2    SYSTERM:
   5  # SRM:
   6    PRINTER:
  45  * SYSTEM45:
Prefix is - SRM:
```

You can see that the system starts out with #5: as the default volume and #45: as the system volume.

Where do the names SRM: and SYSTEM45: come from? They are actually the names of particular directories in the SRM's hierarchy. In this example, the name of the SRM volume is SRM, and the workstation we are using is at node address 45. Since there is a directory SYSTEM45, it is selected as the system volume. All of this selecting is done by the TABLE program as it automatically configures the system each time you boot.

If you need to specify the SRM volume's password, you can do it by using this syntax:

```
SRM:/.<password>
```

The SRM volume password is also the SRM root directory's password. That is, they specify the same thing.

Moving Up and Down the Hierarchy

It would be tedious to type a directory path every time you wanted to access a file. To avoid this, you can specify the **current working directory** using the Filer's Unit Directory command. The current working directory can be used as the "root" to specify subordinate files.

```
U
Set unit to what directory?
#5:/USERS/ROGER Return or ENTER
```

Once you have done this, unit #5: is in effect a volume named ROGER: which contains all the files under directory ROGER in the hierarchy. It's as if you had inserted a disc called ROGER: in a disc drive. If you now command the Filer with:

```
L
List what directory?
ROGER: Return or ENTER
```

it will list all the files in subdirectory ROGER:. You could also use the sequence:

```
L
List what directory?
#5: Return or ENTER
```

since directory ROGER:was installed in #5: by the Filer's Unit Directory command.

Suppose that under ROGER is another directory named MYSTUFF which contains more files. To list the files in MYSTUFF, use the sequence

```
L
List what directory?
ROGER:MYSTUFF Return or ENTER
```

The Filer will realize that MYSTUFF under volume ROGER is itself a directory, and list its contents. If MYSTUFF were not a directory, it would simply be listed as a file in directory ROGER.

You can move the current working directory still farther down the hierarchy in the obvious way. For instance to make MYSTUFF the current directory of #5:

```
(U)
Set unit to what directory?
#5:MYSTUFF (Return) or (ENTER)
```

There was no need to specify the entire pathname from the root, because MYSTUFF was already accessible as a file within volume ROGER.

A special notation is provided to move up the hierarchy. Two periods can be used to denote the "parent" directory of a file. For instance, after moving down to MYSTUFF, unit #5: could be moved back up to the parent directory ROGER by:

```
(U)
Set unit to what directory?
#5:.. (Return) or (ENTER)
```

To go up two levels, use the double-period twice, separated by a slash:

```
(U)
Set unit to what directory?
#5:../.. (Return) or (ENTER)
```

This can be executed all the way up to the root directory. Of course, if you want to get all the way to the top, it is easier to go there directly, using a stroke as the "name" of the root directory. For instance, while #5: is assigned to MYSTUFF you could list all the files in the root directory with the command sequence

```
(L)
List what directory?
#5:/ (Return) or (ENTER)
```

Default Volume vs. Current Working Directory

The current working directory concept is different from the default volume concept. Specifying a current working directory is like installing a disc into a drive unit. Specifying a default volume simply tells the file system what volume name to use when none is specified with a file name.

The two concepts can come together in the Filer's Prefix command. For instance, typing:

```
P
Prefix to what directory?
#5:/USERS/BIG_USER Return or ENTER
```

has two effects since #5: is an SRM unit. The current working directory of #5: is set to /USERS/BIG_USER, and the default volume name is set to BIG_USER. If we now type:

```
U
Set unit to what directory?
#5:.. Return or ENTER
```

the current working directory of #5: becomes USERS (the parent of BIG_USER). However the default volume name is still BIG_USER. So the command

```
L
List what directory?
: Return or ENTER
```

will fail with the message that BIG_USER is not on-line!

The same sort of mistake is commonly made with the system volume. Suppose the current working directory of #45: is SYSTEM45, and the COMPILER, EDITOR and other system files are under SYSTEMS. If the current working directory of #45: is changed, the Operating System won't be able to find the system programs since it thinks of them as SYSTEM45:COMPILER and so on. If this happens, you need to get into the Filer and restore the current working directory of #45:. How can you do so if the Filer is no longer on-line? You will need to execute the Filer by name, specifying a path all the way down from the root to wherever it is:

```
Select (EXECUTE)
Execute what file?
#45:/WORKSTATIONS/SYSTEM/FILER. Return or ENTER
```

Note the dot after the Filer's name. You don't want the system to append .CODE in this case.

Programming With Files

This section describes how to program using the Pascal file operations. It discusses the creation and disposition of files, the basic operations on file data, and the syntax of file names.

Pascal Primitive File Operations

- The following operations put the file into WRITE Mode:

REWRITE	
OPEN	
APPEND	
SEEK	
PUT	
WRITE	
WRITEDIR	
WRITELN	{see the section on TEXT files}
F^	{if the file is already in WRITE Mode}

- The following operations put the file into READ Mode:

RESET	
GET	
READ	
READDIR	
READLN	{see the section on TEXT files}

- The following operations put the file in LOOKAHEAD Mode:

F^	{unless the file was in WRITE Mode}
EOF	{unless the file is open for random access}
EOLN	{see the section on TEXT files}
READ	{of multi-character objects from TEXT files, such as strings, PACs, integers, reals, enumerated types, and booleans.}

Creating New Files

A file is initially created by the REWRITE, OPEN, or APPEND operations. However, OPEN and APPEND are usually applied to existing files. These standard procedures each may take one, two or three parameters:

```
REWRITE (filevar)
REWRITE (filevar,name)
REWRITE (filevar,name,thirdparam)
```

Here "filevar" is the name of a Pascal file variable; "name" is a string which is the system identification of the file; and "thirdparam" is an optional string which is used with Shared Resource Manager files to control shared access to the file (see subheadings "Reset, Rewrite, Open, and APPEND", "SRM Concurrent File Access", and "SRM Access Rights" below). The name string may include information about the file size, and an SRM directory path to the relevant directory.

When a new file is first created, it is considered "temporary", and it will remain so until it is closed with a specification that it be locked into the disc directory. Such temporary files don't conflict with other files of the same name. A new file created by REWRITE, OPEN, or APPEND will be thrown away when the program terminates unless the program takes explicit action.

The allowable file name syntax depends on the Directory Access Method (DAM) being used; this subject is discussed under the preceding section called "File Naming Conventions". However, all file names may have appended to them a specification of the size of the file, which the DAM may use at file creation time to allocate space. The size specification may take the following forms:

- Not present. The file will be allocated the largest available block of space for contiguous-file DAMs (LIF and Workstation 1.0 directory organizations), or an indeterminate amount of space for the SRM. Example: 'CHARLIE.TEXT'
- [*] on end of file name. The file will be allocated the greater of (second largest free block, half of largest free block) for contiguous-file DAMs, or an indeterminate amount of space for the SRM. Example: 'SUSANNAH[*]'
- [nnn] on end of file name, where "nnn" is a positive integer. The file will be allocated nnn blocks of 512 bytes each for contiguous-file DAMs, or an indeterminate amount by the SRM. Example: 'EXACTLY[1000]' which gets 512 000 bytes.

It is permissible to create anonymous files by creating a file without specifying a file name, for example REWRITE(F). Such files will always be placed on the system volume. Note, however, that there is no way to request a specific file size for an anonymous file; REWRITE(F,'[500]') is not acceptable because there is no file name preceding the size specifier.

The REWRITE, OPEN, and APPEND primitives do not necessarily create a new file. Whether they do depends on whether a file already exists with the given name, and whether the file variable is already associated with some physical file by virtue of a previous opening operation.

File Position

In order to understand the three modes a file can be in, we need to take some time to discuss the **file pointer** and the **file buffer, F^**.

Associated with each open file is a file pointer. This pointer can be thought of as a marker indicating how much the file has been read or written. The file pointer starts at the beginning of the file when the file is opened with RESET, REWRITE, or OPEN. The file pointer is set to the end of the file if the file is opened with APPEND. The file element pointed at by the file pointer is called the **current component**. Each time you read from a file, the current component is fetched. Each time you write to a file, the new information becomes the current component.

The components of a file are numbered sequentially from 1 to N, where N is the number of components in the file. The **file position** is a number from 1 to N + 1 which usually corresponds to the position of the file pointer.

The Buffer Variable

Each file has associated with it a special variable called the **buffer variable** or the **file window**. This is a variable of the same type as the components of the file. It is referred to as **F^** where F is the file identifier. For example, if F is a FILE OF INTEGER, then F^ is an integer variable. The buffer variable is usually associated with the current component of the file.

File States

Every file which is open is in one of three states or modes at any given time depending on what was the most recent operation on that file. The file state has to do with whether you are reading or writing the file and whether you have referenced the **buffer variable, F^**. The three state are WRITE Mode, READ Mode, and LOOKAHEAD Mode.

If the file is in WRITE Mode, F^ has no special meaning other than as a variable, and referencing it causes no I/O to take place. This is the mode in which you normally assign to F^, i.e.,

```
  F^ := ... ;
```

in preparation for a PUT statement. If you assign from F^, i.e.,

```
 O... := F^:
```

in this mode you will get unpredictable results.

The READ Mode is also called the LAZY I/O state, because in this mode the buffer varible refers to the current component of the file, but the file system does not fill it until the first time it is referenced. In this mode you normally assign from F^ in order to read the next component of the file.

If the file is in READ Mode, referencing F^ causes the current component to be fetched from the file and placed in the buffer variable. When this is done, the buffer variable is full and the file goes into the LOOKAHEAD Mode. Once the file is in the LOOKAHEAD Mode, F^ may be referenced as many more times as desired but no more I/O will be done.

The LOOKAHEAD Mode is so called because we have **peeked** at the current component without having advanced completely past it. In actuality, the current component has been read into F^ and the file pointer has advanced to the following component. However, the file system pretends that the current component hasn't been fetched yet. In this state the POSITION function returns a value corresponding to the component in the file buffer, which is 1 less than that corresponding to the true file pointer. Also, in this state, READ(F, V) will assign the value of F^ to V instead of reading the next component of the file. On the other hand, if a write were done in this state, it would write the component at the true file pointer, and the POSITION function would appear to advance by 2 instead of 1!

REWRITE(F) (with optional 2nd and 3rd parameters)
If F was already open at the time of REWRITE and no filename is specified, the same physical file is referenced. If a filename is specified, the current file is closed and the physical file specified by the second parameter is referenced. This implicit CLOSE is actually a CLOSE(F,'NORMAL'), and so the file will not necessarily be saved. The file is positioned to its beginning, and any data it contained is discarded. Thus, one way to overwrite the content of an existing file is to open it for reading via RESET, then REWRITE it.

If the file variable F is not already associated with a physical file (that is, F is not presently open), a new file is created and opened for writing. If a file name and size are specified, they will be applied. The new file created is temporary until it is closed, and in fact is distinct from any existing file of the same name.

OPEN(F) (with optional parameters)
Opens a file for random (direct) access, allowing both reading and writing. The file pointer is positioned to the file's beginning.

If F was already open at the time of OPEN and no filename is specified, the same physical file is referenced. If a filename is specified, the current file is closed and the physical file specified by the second parameter is referenced. This implicit CLOSE is actually a CLOSE(F,'NORMAL') and so the file will not necessarily be saved.

If F is not open and no file name is given, an anonymous file is created. If a file name is given matching an existing file, that file is used; otherwise a new file is created.

APPEND(F) (with optional parameters)
If F was already open at the time of APPEND and no filename is specified, the same physical file is referenced. APPEND positions to the end of the file and re-opens it for writing. If a filename is specified, the current file is closed and the physical file specified by the second parameter is referenced. This implicit CLOSE is actually a CLOSE(F,'NORMAL') and so the file will not necessarily be saved. Any data written will get tacked on to the file; the original content remains valid.

If F is not already open and no file name is given, an anonymous file is created and the behavior is like REWRITE command.

If F is not open and a file name is given, APPEND searches for an existing file of that name. If one is found, it positions to the end and prepares for writing; if none is found, it creates a new temporary file.

Restrictions on APPEND:

APPENDing to text files is not allowed in this Pascal implementation. It only works for data files (file of <type>).

If the file is in a volume with a WS 1.0 or LIF directory organization, it may not be possible to APPEND. For this directory type, APPEND is only allowed if there happens to be free space on the disc immediately following the current end of the file.

Disposing of Files

A program terminates the association between a file variable and a physical file with the CLOSE procedure. For example, the call may specify that the file is to be deleted from the directory or made permanent.

CLOSE(F,’SAVE’)
CLOSE(F,’LOCK’)
	Both do the same thing; the file is made permanent in the volume directory. If file is anonymous (has no name), the file is closed and then purged.

CLOSE(F)
CLOSE(F,’NORMAL’)
	Both do the same thing. If the file is already permanent, it remains in the directory. If it is temporary, it is removed.

CLOSE(F,’PURGE’)	The file is removed from the directory whether or not it was permanent.
CLOSE(F,’CRUNCH’)	The end-of-file marker is set at the current file position; data beyond this position is lost. Otherwise like LOCK.

Opening Existing Files

To open an existing file, you must give a file name parameter to the OPEN, APPEND or RESET standard procedures.

RESET(F,’filename’)
Opens an existing file for reading, and positions F to the beginning. If F was already open and no file name is specified, the file to be read is the one which was open. Otherwise, the file system searches for an existing file of the specified name and reports an error if none is found.

RESET(F) with no file name specifier will fail unless F is already open.

OPEN(F,’filename’)
APPEND(F,’filename’)
OPEN and APPEND search for the named file, and if one is found, the association will be with that physical file. But note that if no file is found, a new temporary file will be created (see the comments about file creation shown above).

Note that OPEN(F) and APPEND(F) without a file name will create new files unless F was already open.

REWRITE(F,’filename’)
When REWRITE specifies the name of a file which already exists, a new temporary file is created. All output data goes to this new file instead of the old one. At the time the file is closed using CLOSE(, ’LOCK’) or CLOSE (,’CRUNCH’), the old one is purged and the temporary file is renamed. CLOSE(,’NORMAL’) or CLOSE(,’PURGE’) will purge the new file, leaving the old file intact. This prevents destruction of the old file in case the program terminates prematurely.

To get rid of the old file first, open it with RESET and then do a CLOSE(F,’PURGE’).

Sequential File Operations

In Pascal there are two classes of file: TEXT and DATA. Files of type TEXT are so declared in the Pascal program:

```
VAR F: TEXT;
```

Text files are best thought of as lines of characters, separated by end-of-line designators of some sort. They are intended to represent human-legible text material such as documents.

Data files are files of some component type. They are ordered sequences of variables, all of the same type. The type may be a predeclared type like INTEGER, or some user-declared type:

```
TYPE rec = RECORD
                name: string[50];
                socialsecurity: integer;
            END;
 VAR
    ss: file of rec;
```

A file of char is not the same thing as a text file, because no lines are distinguished in the file of char.

This section is about data files; the discussion of text files is below. In the discussion, F denotes a file variable; T is the type of its components; and V, V1, V2 .. are variables of type T.

READ(F,V)

If F is open for reading (by RESET or OPEN), then this standard procedure will store into variable V the current component of F and advance to the next component. Note that READ(F,V1,V2,V3) is equivalent to three READs in a row. In the LOOKAHEAD Mode, READ (F,V) assigns the value of F^ to V instead of fetching the next component of the file, i.e., no I/O is done.

WRITE(F,V)

If F is open for writing (by REWRITE, APPEND or OPEN) then the value of V is written as the current component of F and F is advanced to the next component. WRITE(F,V1,V2,V3) is allowed.

The file variable name can be referenced as a pointer. It points to the "current" component of the file; that is, if F is a file of T, then F^ is a variable of type T. F^ is called the "buffer variable" of F. (This logical buffer is distinct from the physical device buffer!)

HP Pascal specifies the use of "lazy evaluation", which simply means that the buffer variable is not filled until the program references it.

PUT(F)

PUT and WRITE are related operations. To output data using PUT, first store into the buffer variable the value to be written, then call PUT:

```
F^ := V;
PUT(F);
```

This sequence is equivalent to:

```
WRITE(F,V);
```

Note that it isn't enough to just store into F^; you must also PUT the value. For instance:

```
F^ := V1;
F^ := V2;
PUT(F)
```

will store into the file the single value V2. Also, if you fail to PUT the last component before closing the file, the last component will be lost.

PUT(F) writes the buffer variable, F^, to the current component of the file. That means:

```
F^ := V;
PUT(F);
```

is equivalent to:

```
WRITE(F, V);
```

GET(F)

This is the complementary operation to PUT, used for input. It throws away the current component value and advances the file to the next component.

In WRITE Mode, GET changes the state of the file to READ Mode, but does not change the file position or do any I/O. For example:

```
OPEN(F, 'filename');     {puts file in WRITE Mode}
GET(f);                  {puts file in READ Mode}
V := F^;                 {fetches first file component into V}
```

In READ Mode, GET causes one component to be fetched from the file, which advances the file position by 1, but that component is discarded. For example:

```
RESET(F, 'filename');    {puts file in READ Mode}
GET(F);                  {reads and discards one component}
V := F^;                 {fetches second file component into V}
```

In LOOKAHEAD Mode, GET discards the component in the file buffer, F^, and changes the state of the file to READ Mode. This causes the file position to reflect the true file ponter, thus appearing to advance it by 1. For example:

```
RESET(F, 'filename');   {puts file in READ Mode}
V := F^;                {fetches first file component into V}
GET(F);                 {discards F^ and advances position}
```

which is equivalent to:

```
RESET(F, 'filename');   {puts file in READ Mode}
READ(F, V);             {fetches first file component into V
                            and advances file position}
```

Direct Access (Random Access) Files

Files of DATA (not TEXT) may be accessed directly, that is, a program can specify that it wants to read or write the nth record in the file without scanning through the records in sequence. A file must be opened with the OPEN procedure to allow direct access.

The components of a direct access file are numbered sequentially, with the first being number one. (Note that there is no acknowleged standard in this area; for instance, UCSD Pascal numbers the first component of a direct access file as record zero. All HP Pascal implementations work as described herein.)

When a file is opened, it is positioned at the first component. If sequential I/O operations are performed, the file components will be accessed in ascending order. There are several ways to randomly access the nth record.

READDIR(F,N,V)

The read-direct standard procedure positions F to component N of the file, and then reads the value into variable V. Subsequent READ calls would receive records N+1, N+2 and so on. READDIR(F,N,V1,V2,V3) is equivalent to the sequence

```
READDIR(F,N,V1);
READ(F,V2);
READ(F,V3);
```

Also:

```
READIR(F, N, V);
```

is equivalent to:

```
SEEK(F, N);
READ(F, N);
```

WRITEDIR(F,N,V)

The write-direct procedure positions F^ to component N of the file, and then writes value V. Subsequent writes will place values in components N + 1, N + 2 and so on. For example:

```
WRITEDIR(F,N,V1,V2,V3);
```

is equivalent to:

```
WRITEDIR(F,N,V1);
WRITE(F,V2);
WRITE(F,V3);
```

Also:

```
WRITEDIR(F, N, V);
```

is equivalent to:

```
SEEK(F, N);
WRITE(F, LV);
```

SEEK(F,N)

As with the other direct-access procedures, file F must be opened (for both read and write). SEEK positions F^ so that the next call to READ or WRITE will fetch or place component N.

```
OPEN(F,'CHARLIE');
SEEK(F,100);
GET(F);
V100 := F^;
```

This definition is certainly counter-intuitive in that the program **must not** do an initial GET after opening the file, but **must** after the SEEK command.

SEEK works most smoothly (in the most natural fashion) if used with READ and WRITE:

```
SEEK(F,N);
READ(F,V);
```

Remember that SEEK leaves the file in WRITE Mode, so that in order to read the current component by referencing F^you must first do a GET command. That means:

```
SEEK(F, ;N);
WRITE(F, V);
```

is the same as:

```
SEEK(F, N);
F^ := V;
PUT(F);
```

However:

```
SEEK(F, N);
READ(F, V);
```

is equivalent to:

```
SEEK(F, N);
GET(F);
V := F^;
GET(F);
```

POSITION(F)

This function returns an integer value which is the number of the next component which will be read or written. If the buffer variable F^ is full, POSITION returns the number of that component.

Please be cautious with this function if the file is in the LOOKAHEAD Mode, i.e., if you have read the current component by referencing F^. In this mode, POSITION is correct for reading but is 1 less than the correct value for writing.

MAXPOS(F)

This function returns an integer value which is the number of the last component which has ever been written into the file. Note that the component must have been written; merely SEEKing out to some far component is not enough to cause the maximum position limit to be extended.

Textfile Input and Output

A TEXTFILE is composed of variable-length lines of characters. It differs from FILE OF CHAR in that the lines are variable-length records and are separated by end-of-line marks. Pascal 2.0 and later versions support three different text file representations. Text files are the basis of human-legible input and output. This means that they are used for "formatted" I/O such as printouts.

Declaring a Text File

A text file must normally be declared in the following way:

```
VAR  F: TEXT;
```

All text files must be declared except the two standard files INPUT (corresponding to keyboard) and OUTPUT (which sends its output to the CRT). These two files, if used, must be listed in the main program header:

```
PROGRAM X (INPUT,OUTPUT);
```

However, they must **not** be declared in the body of the program.

In addition, there are two other "standard" system files which may be used, called KEYBOARD and LISTING. If these two files are used, they must appear both in the program heading and in a VAR declaration, as follows:

```
PROGRAM X (INPUT,OUTPUT,KEYBOARD,LISTING);
VAR
 KEYBOARD,LISTING: TEXT;
BEGIN
  ...
END.
```

Don't worry about why INPUT and OUTPUT must not be declared yet KEYBOARD and LISTING must be; that's how it is. Note also that the four standard files are automatically opened by the Operating System before the program runs. The standard files do not generally appear in RESET or REWRITE statements, although they may be closed and re-opened if necessary. Closing and re-opening standard files is **not** recommended.

KEYBOARD and INPUT both take characters from the keyboard; the difference is that characters read from INPUT are echoed to the CRT, while those read from KEYBOARD are not. The file LISTING is opened to PRINTER:LISTING.ASC which is the standard system printer. (Note that since PRINTER: is normally an unblocked volume, the file name part of the specifier is ignored. On the other hand, if PRINTER: is a mass storage volume, the file name is significant. It's a good habit to include a file name even when going to unblocked volumes.)

Representations of a Text File

The way lines of characters will be represented in a text file is determined when the file is originally created.

If the file name given in the REWRITE statement which creates the file ends in the suffix '.ASC', the file representation used is LIF (Logical Interchange Format) ASCII. In this representation, each line is preceded by a signed 16-bit length field telling how many characters are in the line. In this representation, there is no restriction on what characters may appear in the line.

If the creation file name ends in the suffix '.TEXT', the representation used is known as "Workstation 1.0 format". This format is compatible with the UCSD Pascal P-system textfile representation, and may be used as a non-HP interchange format.

The WS 1.0 format precedes lines with a leading-blank compression indication, and terminates each line with an ASCII carriage-return character. Leading blank compression occurs when a line is written, and the compressed blanks are expanded when the line is read. When using this format, don't write the characters NUL (CHR(0)), CR (CHR(13)) or DLE (CHR(16)). Moreover, note that tabs (CHR (9)) are not expanded! Generally it is wise to avoid writing any characters with ordinal value less than 32 into WS 1.0 textfiles.

If the textfile is created anonymously (no file name given) or without a suffix, the "data file" representation is chosen. In this case, a carriage return denotes end-of-line, and all other characters are passed through uninterpreted.

Note

If a file is to be used by the Editor, you should not store control characters (characters with ordinal values less than 32) in it. These characters may cause erroneous cursor placement, which results in data being inserted or deleted in the file at the wrong place.

Note

The representation of a text file is **not** a function of the directory format being used. An ASCII file may be present in a WS 1.0 directory, or a WS 1.0 text file in a LIF directory.

The LIF ASCII representation can only be used if the LIF ASCII Access Method module (ASC_AM) is installed in your system's INITLIB boot file. The WS 1.0 format can only be used if the UCSD Text Access Method (TEXT_AM) module is installed in INITLIB. These modules are present in INITLIB when the Pascal system is shipped, but can be removed if not needed.

If the required Access Method is not installed, the system will choose the "data file" representation regardless of file name suffix.

Formatted Input and Output

The use of WRITE, WRITELN, READ, and READLN for formatted I/O operations with text files is described in many Pascal reference documents and will not be repeated here, except to take note of the behavior when reading and writing character strings.

HP Pascal supports two forms of character string, generically referred to as PAC (for Packed Array of Char) and STRING. A PAC is a variable whose type specification is of the form

```
TYPE  T = PACKED ARRAY [1..N] OF CHAR;
```

where N is some integer constant. The lower bound of a PAC must be 1 in HP Pascal, although Series 200 Workstation Pascal allows any arbitrary lower bound if the $UCSD$ Compiler option is used.

When a string literal value is assigned to a PAC, if the string is shorter than the declared length then the string is blank-padded to the declared length. Thus if a 5-character literal is assigned to a 10-character PAC, the last 5 characters of the PAC will get blanks. This same behavior occurs on input of a PAC value (see below).

When a PAC is written to a text file, all N characters are put out unless a shorter field specification is given in the WRITE statement:

```
TYPE
 PAC = PACKED ARRAY [1..10] OF CHAR;
VAR
 S: PAC;

S := 'abcde';        {PAD WITH 5 TRAILING BLANKS}
WRITE(F,S);          {WRITE 10 CHARACTERS}
WRITE(F,S:5);        {WRITE FIRST 5 CHARS}
WRITE(F,S:15);       {WRITE 5 BLANKS, THEN ALL 10 CHARS OF PAC}
```

A STRING is a variable whose type specification is of the general form:

```
TYPE  S = STRING [N];
```

where N is a constant between 1 and 255 giving the maximum allowable length of the string. STRINGs differ from PACs in having an implicit variable "current" length. Usually the length of a string is the length of the last string value assigned to it, although string length can be explicitly manipulated by the standard procedure SETSTRLEN.

When a STRING variable is read from a text file, its length is set to the length of the incoming string (see below). When written, a STRING takes the number of characters specified by its current length.

Reading a String or PAC from a Textfile

When a string is read from a textfile, its length is usually determined by an end-of-line marker.

If the entire string is filled before end-of-line is reached, the read operation ceases. No error is reported, and the next character read will be the one following the last one read.

When reading strings, an end-of-line must be explictly passed by READLN. If you repeatedly read into a string while positioned at an end-of-line marker, you will keep getting back an empty STRING or a PAC of all blanks. The approved way to read long lines into short strings is:

```
WHILE NOT EOF(F) DO
 BEGIN
  REPEAT
   READ(F,S);
   ...  {process the piece of string}
  UNTIL EOLN(F);
  READLN(F);
  ...
 END;
```

You should be aware of one other fact about end-of-line handling in READs: reading strings or PACs is the only situation in which end-of-line is not automatically "swallowed". The Standard states that when EOLN(F) is true, the value of F^ is a blank. When reading a number, for instance, end-of-line is not treated differently from any other blank in the character stream of the input text file.

RESET, REWRITE, OPEN and APPEND

The optional third parameter to the standard file opening procedures is used at the time of file creation to control concurrent access to files and to specify file access rights via passwords. This parameter is a character string whose syntax conforms to the following definition:

```
third_param          ::=  [ concurrency_word ]
                     ::=  [ password_list ]
                     ::=  concurrency_word "," password_list

concurrency_word     ::=  "SHARED"
                     ::=  "EXCLUSIVE"
                     ::=  "LOCKABLE"

password_list        ::=  capability [ ";" capability ]
capability           ::=  password ":" access_right_list
access_right_list    ::=  access_right { "," access_right }

access_right         ::=  "READ"
                     ::=  "WRITE"
                     ::=  "PURGELINK"
                     ::=  "CREATELINK"
                     ::=  "SEARCH"
                     ::=  "MANAGER"
                     ::=  "ALL"
```

Note that in the passwords themselves, upper and lower case letters are distinct. Examples of third parameter strings:

```
'SHARED'
'EXCLUSIVE,MYSECRET:MANAGER'
'LOCKABLE,R:READ;W:WRITE'
'Charley:ALL'
```

Debugging Programs Which Use Files

The file system uses the TRY-RECOVER and ESCAPE mechanism in its normal internal operations. For instance, when opening a file several escapes may occur intemal to the File System or driver calls. These "errors" don't get back out to the user program.

But if the Debugger is used on such a program and error trapping is enabled, the Debugger will stop the computer on each internal escape. This can be very confusing. The clue that this is happening is that the line number displayed by the Debugger in the lower right corner of the screen doesn't change during the FS call.

The most common escape codes generated in this fashion are -10, 2080 and -26. You can suppress the Debugger's activity on these codes with the following "Escape Trap Not" Debugger command:

```
ETN -26 2080 -10
```

SRM Concurrent File Access

Three modes of access to shared files are allowed:

- EXCLUSIVE: No concurrency. Only one workstation may open the file at a time. This is the default for all files opened on the SRM.
- SHARED: No controls. The file may be opened by any number of workstations for both reading and writing. This is particularly dangerous for multiple writers since, for performance reasons, some local buffering is done in each workstation. Different buffers may overlap parts of the same file, and may not contain identical data! Shared file users will not be aware of changes in actual end-of-file induced by the actions of other users.

  Shared files are primarily intended to be used by multiple readers.
- LOCKABLE: This mode provides strict concurrency interlocking by means of file operations LOCK, WAITFORLOCK, and UNLOCK. The file must be locked to perform any operation on it; only one reader/writer may access the file at a time. A series of operations or a single operation be be performed while it is locked. The initial lock obtains the necessary physical file status information from the SRM, and unlocking updates all the status information on the SRM as well as flushing buffers. Thus when the file is unlocked, its contents are always complete and consistent.

Note

Shared access is allowed concurrently with lockable access and may circumvent the integrity provided by the locking mechanism.

The user-callable routines which support locking are provided in the library module LOCKMODULE, which is in the standard system LIBRARY. To use them, the program must IMPORT LOCKMODULE. The specifications for these routines are:

- FUNCTION LOCK (ANYVAR F:FILE): BOOLEAN; This function returns true if the lock succeeded, or false if the lock failed because the file was already locked. Other IO errors such as file not open generate an error which may be trapped using TRY/RECOVER (see System Programming Language Extensions).
- PROCEDURE WAITFORLOCK (ANYVAR F:FILE); This procedure sends the SRM a request to lock the file, and then waits until it is confirmed.
- PROCEDURE UNLOCK (ANYVAR F:FILE); This procedure releases the file so another workstation can lock it.

The file locking capabilities are primarily intended for data files (Pascal file of <type>) which are opened for random access using the standard procedure OPEN. Suppose F is a file which is not already open. The cases are:

- `OPEN (F, 'filename')` The existing file is opened for exclusive access. The open will fail if the file is already open by some other workstation. This is the default.
- `OPEN (F, 'filename', 'EXCLUSIVE')` The existing file is opened for exclusive access. The open will fail if the file is already open by some other workstation.
- `OPEN (F, 'filename', 'SHARED')` The file is opened for shared access. Any number of workstations may have the file open SHARED at the same time. They may read or write — there is no synchronization.

- `OPEN (F, 'filename', 'LOCKABLE')` The file is opened in such a way that no access is permitted unless the file is first put in the locked state. Any number of workstations may have a file open LOCKABLE at a time, but only one workstation may have the file locked.

REWRITE, to a file which is already open within the program performing the REWRITE, simply repositions the file to its beginning and sets it up for writing.

If REWRITE specifies the name of a file which does not exist, a new file of that name is created and used.

If a physical filename is given and a file of that name exists, the existing file is opened with whatever concurrency specification (shared, exclusive) was given in the REWRITE. If no physical file exists, one of the given name is created and opened with the requested concurrency specification. This action is in addition to the creation of the temporary file, and helps prevent interference by other workstations.

Surprising effects may occur if two workstations REWRITE the same physical file concurrently. The one closed last will remain in the SRM directory.

Note that `REWRITE(F,'LOCKABLE')` is probably not a sensible operation. However, it does not generate an error.

SRM Access Rights

Passwords can be used to restrict the types of access allowed to a file (on the SRM a directory is also a file). They can be set by the Filer's Access command, or at the time a file is created. Passwords can control the following six types of access:

- READ
- WRITE
- SEARCH
- CREATELINK
- PURGELINK
- MANAGER
- ALL

Any access rights for which no password is specified belong to the set of public capabilities which are granted to any workstation opening the file without specifying passwords.

The word ALL denotes the six access types collectively. When an ALL password exists, there are no public capabilities. The ALL password allows any file operation to be performed.

SEARCH capability is required on all directories along the pathname to a given file.

RESET requires READ access to the file.

Both READ and WRITE capability are required if the file is opened by calls to OPEN or APPEND.

To REWRITE an existing file, any passwords in the file specification (second parameter to REWRITE) are used only to purge the old file. However, one of the three capabilities READ, WRITE, MANAGER must also be granted to open the old file before purging it. The new file created by REWRITE will have the passwords specified in the third parameter; until this new file is closed, any operations may be performed on it.

WRITE capability on the directory in which it resides is required to CLOSE 'PURGE' a file, in addition to the SEARCH capability needed to open the file and PURGELINK capability on the file.

To CLOSE 'LOCK' a file, WRITE capability is required for the directory, in addition to the SEARCH capability needed to open the file.

If a password with MANAGER capability is used to open a file, any file operations may be performed, since the manager password would allow the access types to be changed. For example,

`REWRITE(F,'FILE1','A:ALL')` Gives no public capabilities.

`REWRITE(F,'FILE1','M:MANAGER')` All capabilities except MANAGER are public. This allows any file operations to be performed, but the manager password 'M' is required to change or set passwords.

The Editor

Chapter 3

Introduction

This chapter introduces the features of the Pascal Workstation's Editor. The Editor enables you to create, change, store and retrieve both programs and textual documents. The Editor has built-in reminders (prompts) and uses single keystroke commands.

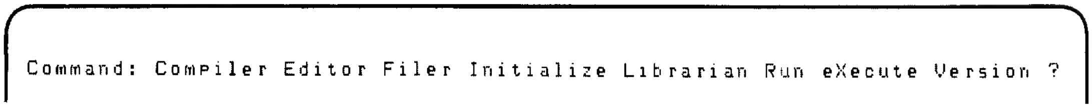
The Editor is a cursor-based screen editor. (The cursor is the blinking underline symbol on the screen). This provides access to any part of a text file through movement of the cursor. You can move rapidly through text files to read and edit your text.

The programs and documents created by the Editor are usually stored as TEXT files but can be stored as ASCII files or DATA files. ASCII files can be used with the other language systems that run on your computer.

This chapter has four main sections. The first two sections demonstrate how to enter and use the Editor by leading you through writing a short Pascal program. The next section, ''A Closer Look'', presents more detailed information about the Editor. The last section, ''Editor Commands'', contains an overview or summary of all the Editor commands, a glossary of terms, and a semantic and syntactic description of each Editor command in alphabetical order. If you have questions about any of the commands covered in the sample session, this last section should answer them. Once familiar with the Editor, you can use the overview/summary of the Editor commands for quick reference.

Entering the Editor

It is assumed that the Pascal System is already "up and running".



```
Command: Compiler Editor Filer Initialize Librarian Run eXecute Version ?
```

This prompt tells you that you are at the system's Main Command Level – the level from which all the Pascal subsystems (Compiler, Editor, Filer, etc.) are entered. Entry to any subsystem is accomplished by typing the first character of the name of the subsystem you wish to enter.

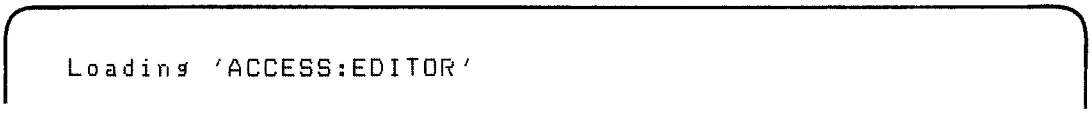
Note

If you have a system workfile (created in a previous Editor session or in the Filer subsystem), first go into the Filer and use the Save, New and Quit commands to store and release the workfile. Then exit the Filer subsystem.

When the system is delivered to you, the Editor is on the disc labeled "ACCESS:" and is named:

```
ACCESS:EDITOR
```

Now press the E key. You can use uppercase or lowercase: the computer treats both exactly the same while at the Main Command Level. If the Editor code file is on-line, the screen clears and displays:

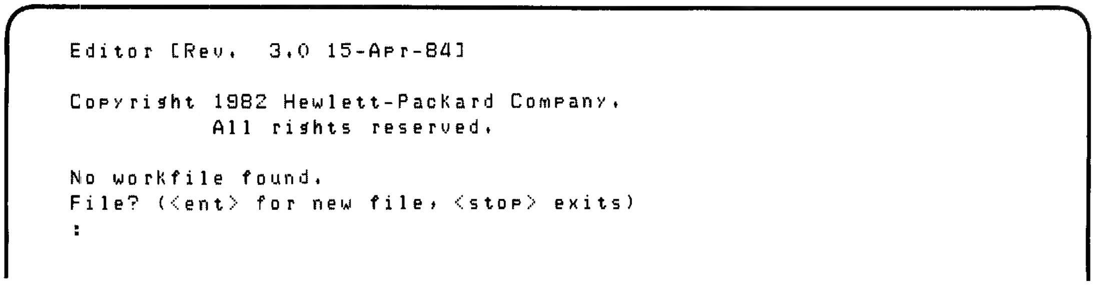


```
Loading 'ACCESS:EDITOR'
```

If you copy the Editor code file to another disc, which has a different volume name, you should tell the operating system where to look for the Editor. (See the What command in Chapter 1)

Creating a Text File

When you enter the Editor, the following prompt is displayed.



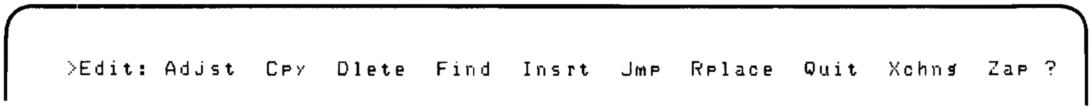
```
Editor [Rev. 3.0 15-Apr-84]

Copyright 1982 Hewlett-Packard Company.
         All rights reserved.

No workfile found.
File? (<ent> for new file, <stop> exits)
:
```

This tells you that you are entering the Editor without a system workfile and requests a file name. Respond by pressing the Return or ENTER key to instruct the Editor to create a new text file for your use. The file will be named later when leaving the Editor.

The Editor can also be directly entered from the Compiler subsystem. This is covered in the Compiler chapter.

The Editor Prompt

The screen clears again and displays the Editor prompt on the top line:



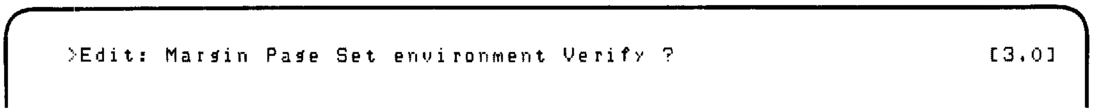
```
>Edit: Adjst  Cpy  Dlete  Find  Insrt  Jmp  Rplace  Quit  Xchng  Zap ?
```

You are now in the Pascal Editor with a new file. The Editor prompt shows the most common commands used in the Editor. This is called a "prompt" because it prompts you to take some action, i.e., give the Editor a command.

The first character of the prompt line (> or <) indicates the direction of cursor movement (i.e., the way the cursor moves when TAB, Return or ENTER keys, or the space bar is pressed). When the ">" character is displayed, the cursor will move "forward" in the text. Similarly, when the "<" character is displayed, the cursor will move "backwards" in text. Pressing >, ., or + will set forward direction, while <, ,, or - will set reverse direction.

The character indicates the direction that searches take place in the Find and Replace commands, also the Delete and Page commands.

The prompt line shows a partial list of commands available in the Editor. To see the rest of the commands, type ?. The screen shows the Editor's alternate prompt:



```
>Edit: Margin Page Set environment Verify ?                [3.0]
```

This alternate prompt also shows the revision number of the Editor enclosed in brackets. Type ? again and the main Editor prompt reappears.

All of the commands in the Editor are initiated by typing a single key corresponding to the first character of the command shown in the Editor prompt. Again it does not matter whether the character is uppercase or lowercase — the Editor accepts either one. Now that you are in the Editor and understand something about the Editor prompt, let us begin the sample Editor session.

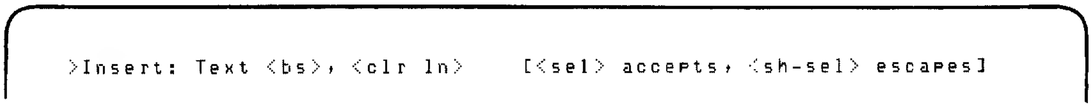
A Sample Editor Session

Feel free to skim this section if you are familiar with screen oriented editors. You may even prefer to try out the Editor commands on your own. If you choose to experiment with the Editor commands, do not use any files you cannot afford to lose.

If you are still reading, step through the following examples on your machine. Doing the examples will help you learn faster than just reading about them.

Creating Text

The most direct way to generate text is with the Insert command. Initiate the Insert command by pressing I and the screen responds with the following prompt:



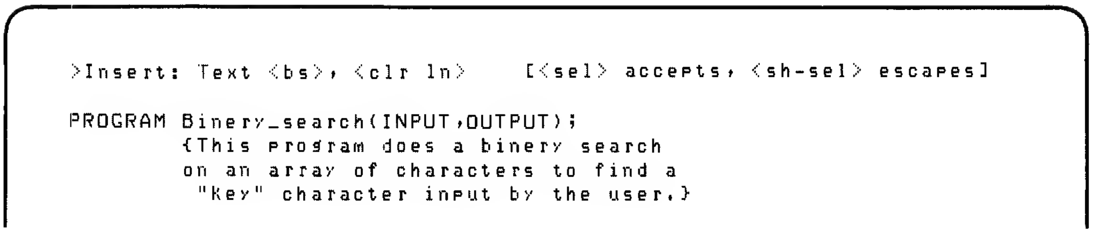
```
>Insert: Text <bs>, <clr ln>    [<sel> accepts, <sh-sel> escapes]
```

While in the Insert command, any of the regular character-entry keys (the main keyboard) or the numeric pad keys (on the right) may be used. With a few exceptions, using the key clusters on the top of your keyboard or CTRL key sequences is not advised. (Most of these keys generate a question mark (?) while in the Insert command. Others have results which may surprise you). Use CTRL key sequences only if you are working with Stream files. (See Chapter 1 for details on the use of Stream files). The exceptions are the cursor control keys, BACK SPACE, CLR LN, ANY CHAR and DUMP ALPHA (which sends a copy of the screen image to your printer).

Let's start typing in a Pascal program now. Press Return or ENTER and then type the text shown in the following display. If you make a mistake, use BACK SPACE to move the cursor backward and then type the correction. You can use CLR LN to delete the most recently inserted line. Prompts in the Editor always show actual key options in the form of a key abbreviation shown in (<) and (>) symbols.

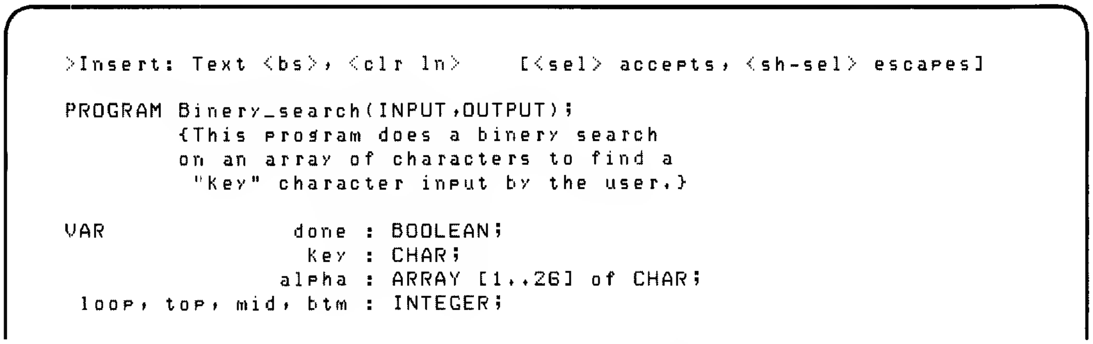
The word "binary" is misspelled in the display; leave it that way for now.

Notice that when you press Return or ENTER after typing the first line, the cursor returns to column one (the column that the "P" in PROGRAM is in). To type the second line, use TAB to indent the comment enclosed in the braces. The next time you press Return or ENTER the cursor automatically returns to the indented position created in the previous line. This indenting feature proves handy when writing Pascal programs as it adds visual clarity to the code.



```
>Insert: Text <bs>, <clr ln>    [<sel> accepts, <sh-sel> escapes]

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "Key" character input by the user.}
```

The display above shows what your screen should look like after the first few lines are typed. To move the cursor back to column one for the next line, press and hold (BACK SPACE). The keyboard automatically "repeats" any key that remains pressed.



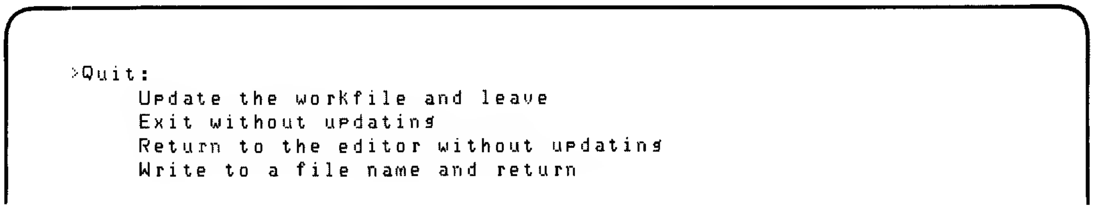
```
>Insert: Text <bs>, <clr ln>    [<sel> accepts, <sh-sel> escapes]

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "Key" character input by the user.}

VAR             done : BOOLEAN;
                 Key : CHAR;
               alpha : ARRAY [1..26] of CHAR;
 loop, top, mid, btm : INTEGER;
```

When your screen looks like the display above, press (Select) ((EXECUTE)) to complete the insertion. The screen displays the Editor prompt along with the text you inserted. Next we will save this program fragment on the disc and then return to create more text.

Storing Your File and Returning to the Editor

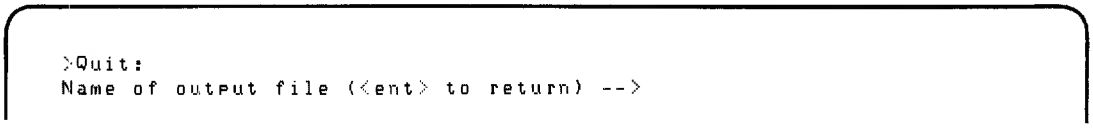
This section shows how to save a file on a disc and then return to the Editor. It is a good idea to do this periodically when writing and editing large text files. Although power outages occur infrequently, it can be devastating to lose an entire session of work. Occasional updating of your file secures your work against this possibility.

Press (Q) to initiate the Quit command. The screen clears and displays:



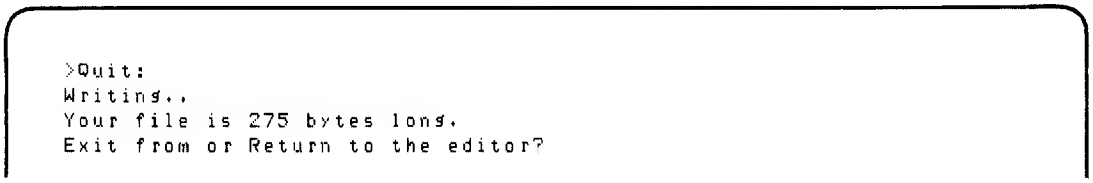
```
>Quit:
     Update the workfile and leave
     Exit without updating
     Return to the editor without updating
     Write to a file name and return
```

Before typing anything, find the disc labeled `DOC:` and insert it in your disc drive in place of the disc labeled `ACCESS:`. Now press (W) and the screen displays:



```
>Quit:
Name of output file (<ent> to return) -->
```

The prompt is requesting a file specification. Respond by typing DOC:BINSEARCH followed by Return or ENTER. The screen now displays:



```
>Quit:
Writing..
Your file is 275 bytes long.
Exit from or Return to the editor?
```

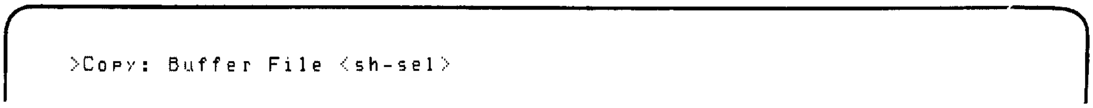
The exact number of bytes may differ with what is indicated in the line above.

Now press R. The screen fills with your text and the cursor is positioned where it was when you initiated the Quit command.

Copying Text from Other Files

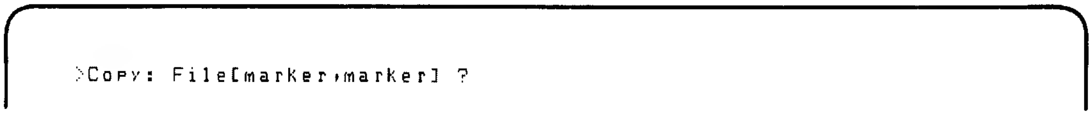
The Insert command is the most common way of creating text but other commands are available. The Copy command allows you to copy specified text from another file.

On the DOC: disc is a text file called BINDOC.TEXT which you are going to copy into your current text file. Position the cursor by pressing J and then E (for Jump to End). This command sequence moves the cursor to the end of your text file. (More on the Jump command later). Now press C and your screen displays:



```
>Copy: Buffer File <sh-sel>
```

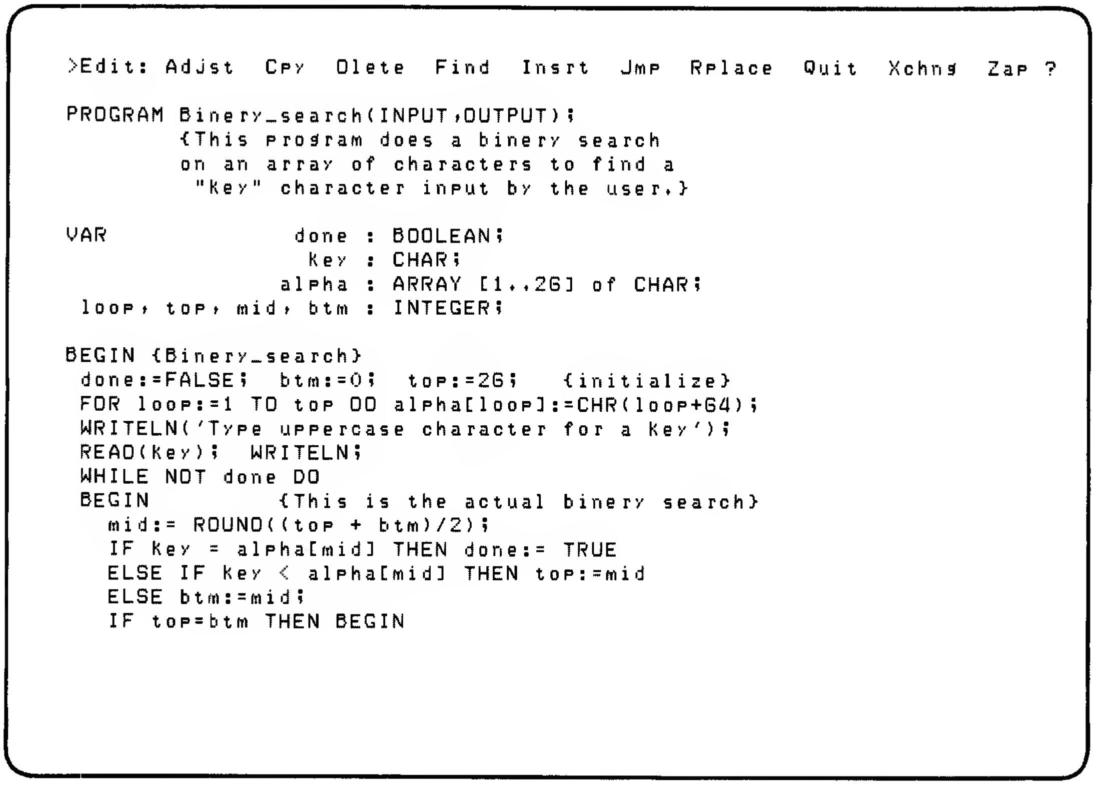
The Buffer option is demonstrated along with the Delete command later in this section. Now press F (to Copy from a File) and the new prompt appears:



```
>Copy: File[marker,marker] ?
```

The system is requesting a file specification. Type DOC:BINDOC and press Return or ENTER. The .TEXT part of the file name does not have to be typed; it is automatically supplied by the Editor. The volume name , DOC:, had to be specified because otherwise the Editor would look for the file on ACCESS:, the default volume. See Chapter 2 for further information on the default volume.

The entire file DOC:BINDOC.TEXT has been copied into your current text file in memory. The copy always occurs at the cursor position. This is why you moved the cursor to the end of the file before the copy. The screen now appears as follows:



```
>Edit: Adjst  Cpy  Dlete  Find  Insrt  Jmp  Rplace  Quit  Xchng  Zap ?

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "Key" character input by the user.}

VAR               done : BOOLEAN;
                   Key : CHAR;
                 alpha : ARRAY [1..26] of CHAR;
 loop, top, mid, btm : INTEGER;

BEGIN {Binery_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a Key');
 READ(Key);  WRITELN;
 WHILE NOT done DO
 BEGIN         {This is the actual binery search}
   mid:= ROUND((top + btm)/2);
   IF Key = alpha[mid] THEN done:= TRUE
   ELSE IF key < alpha[mid] THEN top:=mid
   ELSE btm:=mid;
   IF top=btm THEN BEGIN
```

To Copy only part of a file, a beginning and ending marker are specified. These markers must have been previously set in the text file being copied. (See the Set command in the "Editor Commands" section of this chapter for further information on setting markers). Now that you have your screen full of text, let's look at the general pattern of leaving an Editor command and some ways to move the cursor.

Confirming or Aborting Commands

The (Select) ((EXECUTE)) key tells the Editor to accept all of the insertions or changes you have made in the text file. The cursor remains where it was when you pressed (Select). Conversely, holding down the (SHIFT) key while pressing (Select) (shown as (SHIFT)-(Select)) tells the Editor to ignore all of the changes made since initiating the command and leaves the cursor where it was when the command was initiated. (The (ESC) key also performs this function on keyboards so equipped.) Both key sequences ((Select) and (SHIFT)-(Select)) return you to the main Editor prompt.

The changes are stored in the computer's internal read/write memory but are not made permanent on a mass storage medium until you exit the Editor and use one of the options that writes the information to a file.

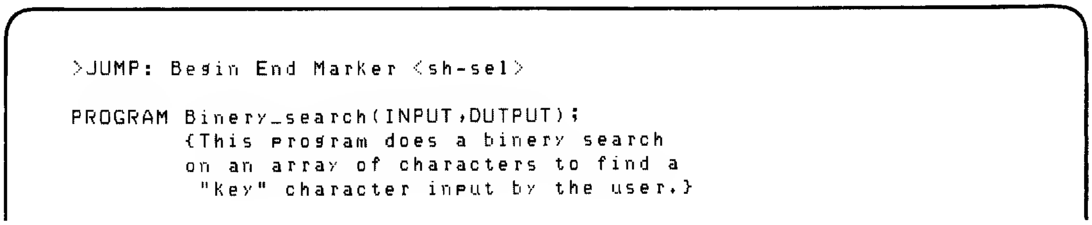
Not all commands let you abort changes with SHIFT-Select and not all require Select for acceptance. For instance, the Copy from buffer command is accomplished by simply pressing C B. The text is copied and the Editor's prompt appears with no other action on your part. The specifics of how each command uses these keys is discussed as each command is presented.

Moving the Cursor

Now that you have some text on the screen, experiment with positioning the cursor. The arrow keys, the Return and ENTER keys, the Tab key, the space bar, the mouse, and the cursor wheel (also called the knob) all move the cursor. The wheel normally moves the cursor left or right, depending on which direction you turn it. If you hold down the Shift key while turning the wheel, the cursor moves up or down while remaining in the same column position.

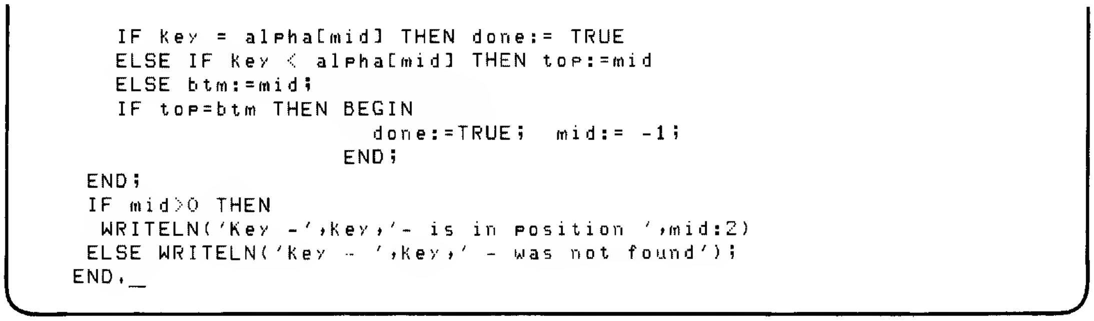
An integer in the range 1 to 9999 can be used as a "repeat factor" before all of the cursor control keys and some of the Editor commands. (Repeat factors must be in the range 1 to 4095 for use with the TAB key). The result will be the same as if you had pressed the key that many times. For instance, typing the number 42 and then pressing the space bar causes the cursor to move 42 characters in the current direction.

The Jump Command offers another means of cursor positioning. Press J and the top of your screen displays:



```
>JUMP: Begin End Marker <sh-sel>

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "Key" character input by the user.}
```

Typing B causes the cursor to jump to the beginning of the file, in this case directly above the (P) in PROGRAM, and the Editor's main prompt reappears. Now press J then E and the cursor moves to the end of your text file as shown in this partial display:

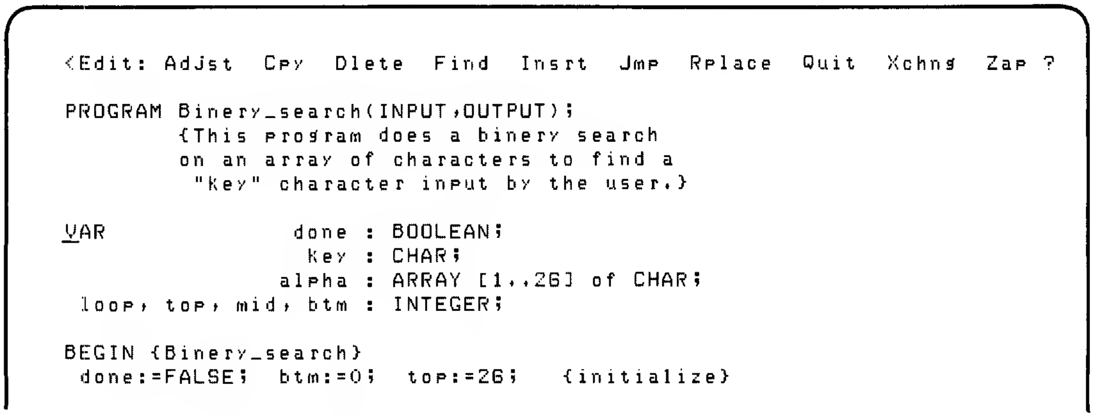


```
     IF Key = alpha[mid] THEN done:= TRUE
     ELSE IF Key < alpha[mid] THEN top:=mid
     ELSE btm:=mid;
     IF top=btm THEN BEGIN
                       done:=TRUE;  mid:= -1;
                     END;
   END;
   IF mid>0 THEN
    WRITELN('Key -',Key,'- is in position ',mid:2)
   ELSE WRITELN('Key - ',Key,' - was not found');
  END._
```

You can also Jump to previously set markers (see the Set command in the "Editor Commands" section) by typing J M followed by a marker name.

The beginning and end of a file are simply the first and last characters in the current text file. In this case, the position directly above the (P) in PROGRAM and the space following the final word END. are the first and last characters, respectively. The Editor adjusts these internal pointers automatically as the text file is changed.

The Page command lets you move through a file one screen (23 lines) at a time. It is roughly equivalent to using a repeat factor of 23 with ↑ or ↓ depending on the direction shown in the prompt. If the cursor is not at the end of the file, press J E. Now type < to change from the forward to the backward direction and press P (for Page). The top half of your screen now looks like:



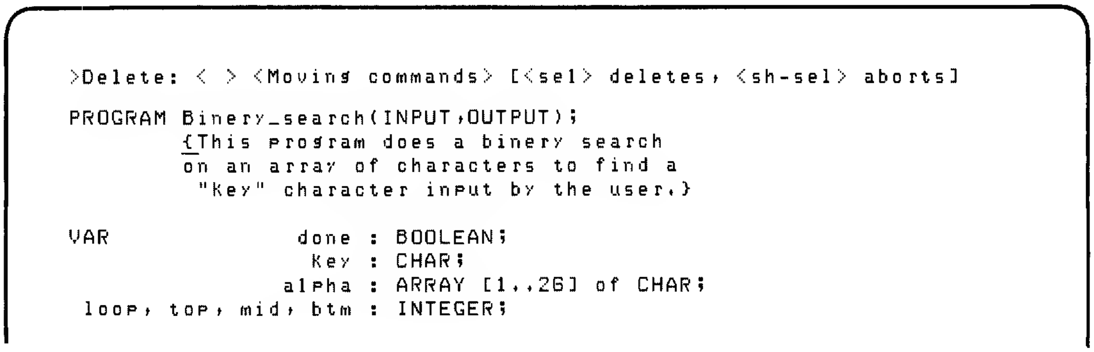
```
<Edit: AdJst  Cpy  Dlete  Find  Insrt  Jmp  Rplace  Quit  Xchng  Zap ?

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "Key" character input by the user.}

VAR            done : BOOLEAN;
                Key : CHAR;
              alpha : ARRAY [1..26] of CHAR;
 loop, top, mid, btm : INTEGER;

BEGIN {Binery_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
```

Notice that the cursor is positioned at the VAR declaration in the program which is 23 lines from the end of the file. Since the cursor movement direction is still backward, type > to change it to forward. The Page command is especially handy when moving through a large file.

Deleting Text

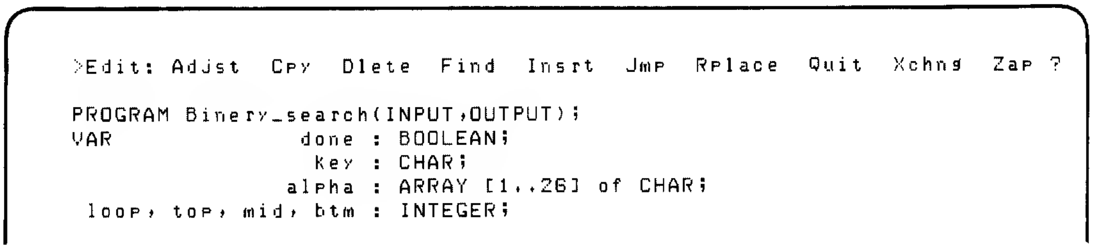
Now position the cursor under the first bracket on the second line of the program and press D. This initiates the Delete command. Moving the cursor removes text from the file. To restore the text, use any cursor control key which moves the cursor back over the area where text has been removed. The BACK SPACE key and the cursor wheel work well for this.

Upon pressing D, the screen displays:



```
>Delete: < > <Moving commands> [<sel> deletes, <sh-sel> aborts]

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "Key" character input by the user.}

VAR            done : BOOLEAN;
                Key : CHAR;
              alpha : ARRAY [1..26] of CHAR;
 loop, top, mid, btm : INTEGER;
```

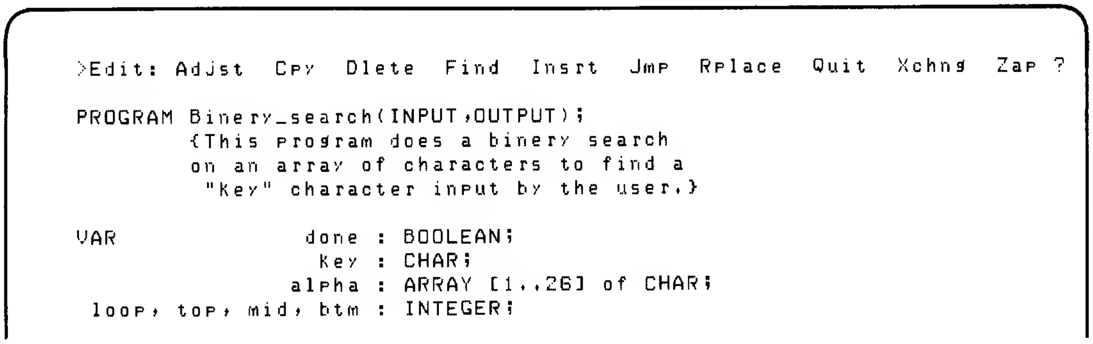
First make sure the direction is forward (>) as shown above and then type 4 followed by Return or ENTER. This uses a repeat factor and moves the cursor 4 lines, deleting text as it goes. (The deleted text is temporarily stored in the copy buffer). Now press Select (EXECUTE) and the screen displays:



```
>Edit: Adjst  Cpy  Dlete  Find  Insrt  Jmp  Rplace  Quit  Xchng  Zap ?

PROGRAM Binery_search(INPUT,OUTPUT);
VAR              done : BOOLEAN;
                  key : CHAR;
                alpha : ARRAY [1..26] of CHAR;
 loop, top, mid, btm : INTEGER;
```

Before typing any other keys or moving the cursor, press C then B. This takes the information stored in the copy buffer and copies it into the text file beginning at the current position of the cursor. Since the Delete command just filled the buffer with the text that was removed, the Copy from Buffer command simply returns the screen to its state before the Delete command was entered.

The top of the screen should now display:



```
>Edit: Adjst  Cpy  Dlete  Find  Insrt  Jmp  Rplace  Quit  Xchng  Zap ?

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "key" character input by the user.}

VAR              done : BOOLEAN;
                  key : CHAR;
                alpha : ARRAY [1..26] of CHAR;
 loop, top, mid, btm : INTEGER;
```

Recovering Deleted Text

As the example shows, even if you complete the Delete command using Select (EXECUTE) instead of SHIFT-Select, you can still change your mind and recover that text using the Copy (from buffer) command. Take care not to wait too long or depend on this too heavily as there are other commands which alter the contents of the buffer. None of the cursor movements alter the buffer in any way.

Moving and Duplicating Text

The sequence of the Delete and Copy (from buffer) commands provide a convenient way of moving text to different parts of the file. For instance, in the operation just completed above, any of the cursor control keys could have been used to reposition the cursor after the deletion occurred and before the Copy from the buffer was executed.

The buffer is "filled" with the text affected by the Delete command and by the Insert and Zap commands. Doing a Copy from buffer sequence does not change the contents of the buffer. This feature lets you copy the same text in numerous places.

Whether the Delete command was completed with the Shift Select (EXECUTE) or Select (SHIFT-EXECUTE) sequence makes no difference to the copy buffer. What this means in practical terms is that the Delete command allows you to fill the buffer without affecting your original text.

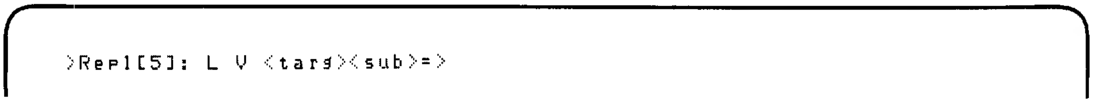
So if you want to duplicate the text instead of moving it to a different location, use the sequence:

1. Press D to initiate the Delete command.
2. Cause some cursor movement. This deletes text and stores it in the copy buffer.
3. Press SHIFT-Select to recover the text that was just deleted.
4. Reposition the cursor to where you want to duplicate the text.
5. Press C B to execute the actual copy at the new cursor position.

Changing and Altering Text

Mistakes or necessary changes in a program or text file are not always obvious when creating text. The Editor features commands which allow you to go back and make changes when needed. These are the Replace and eXchange commands and the Delete/Insert sequence. These will be demonstrated by making corrections to the sample program text.

Press J B to move the cursor to the beginning of the file and then type 5 and press R to initiate the Replace command. The prompt at the top of the screen appears:



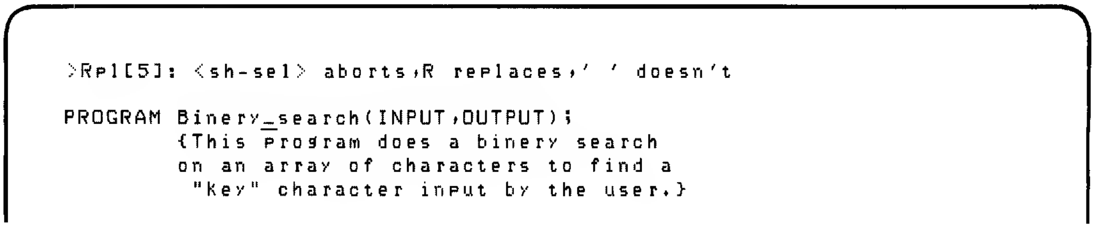
```
>Repl[5]: L V <targ><sub>=>
```

Press L and V to tell the Editor that you are going to give it a Literal string and that you want to operate in the Verify mode. A Literal string may occur as a word or as part of a word. The alternative is a Token string which must occur as a word. The Verify mode makes the changes one at a time after asking you if you want this occurrence replaced. Now type:

```
/inery//inary/
```

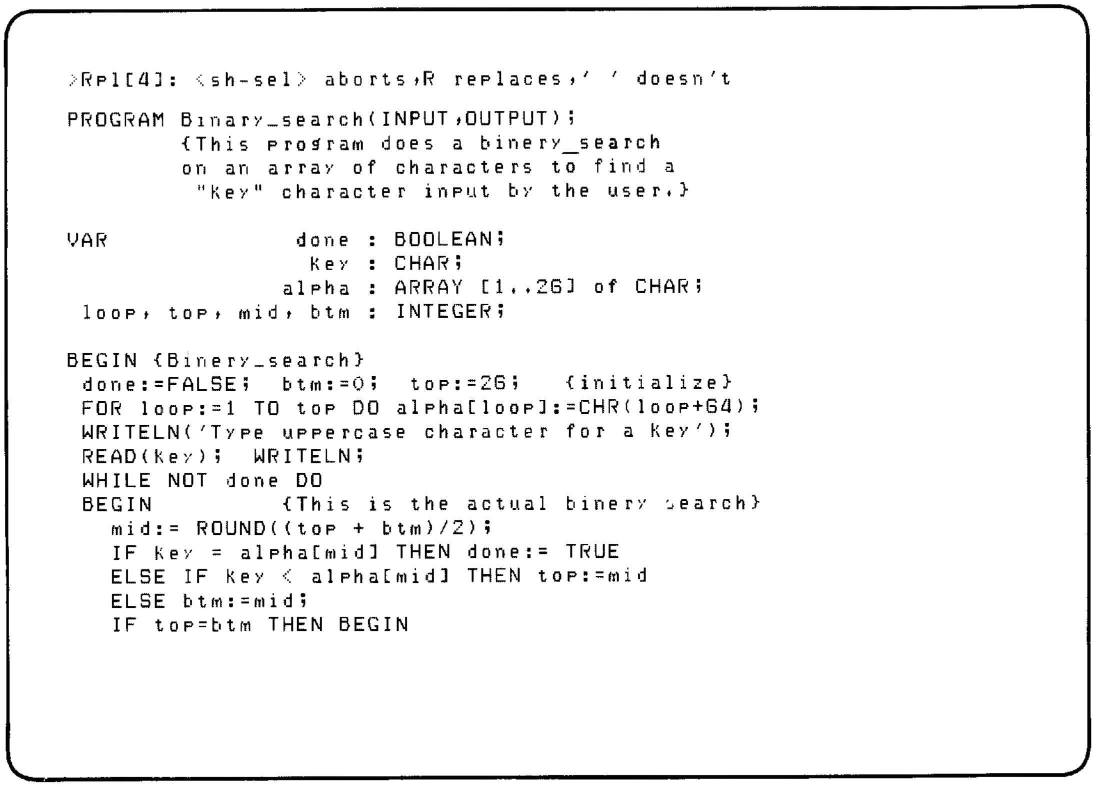
The slashes are used to delimit the target and substitution strings. Any non-alphanumeric or non-control characters can be used as delimiters. This is necessary when the slash is part of the search string or replacement string. Notice also that "inery" is specified instead of "binery". This is because two occurrences of the word are "Binery". The two words are unequal.

After you type the final delimiter, the screen clears and displays:



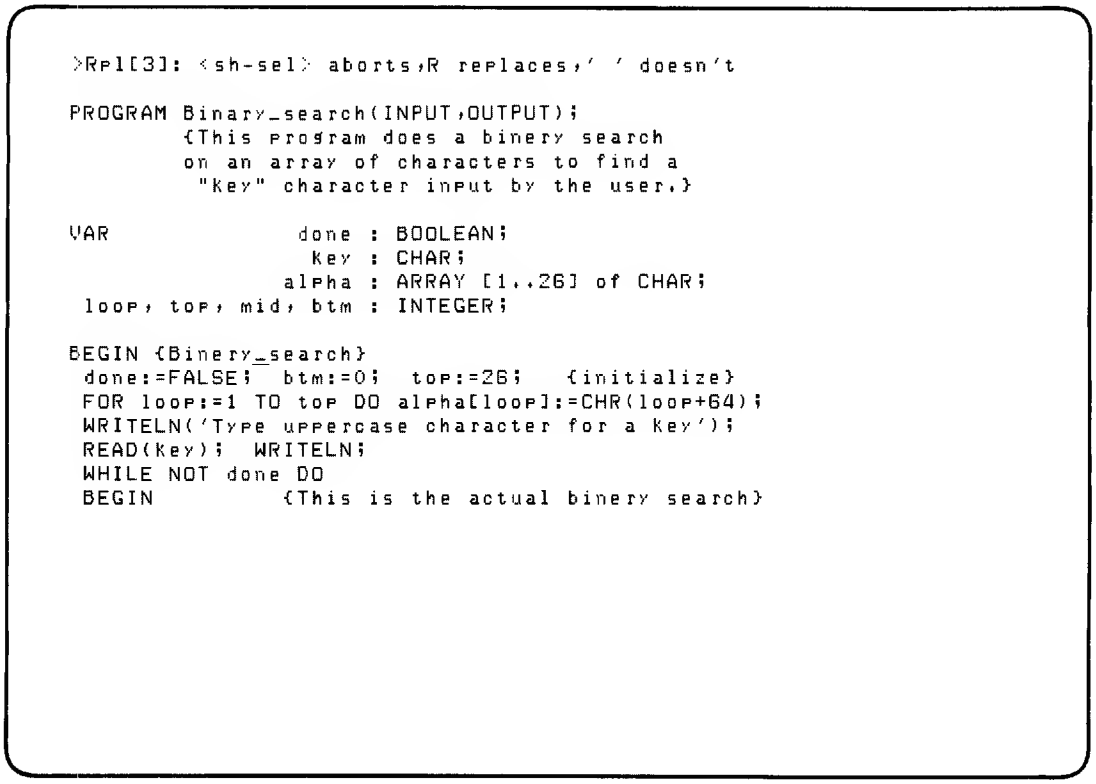
```
>Rpl[5]: <sh-sel> aborts,R replaces,' ' doesn't

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "key" character input by the user.}
```

The cursor is positioned behind the first occurrence of the string `inery`. Now press R and watch what happens:



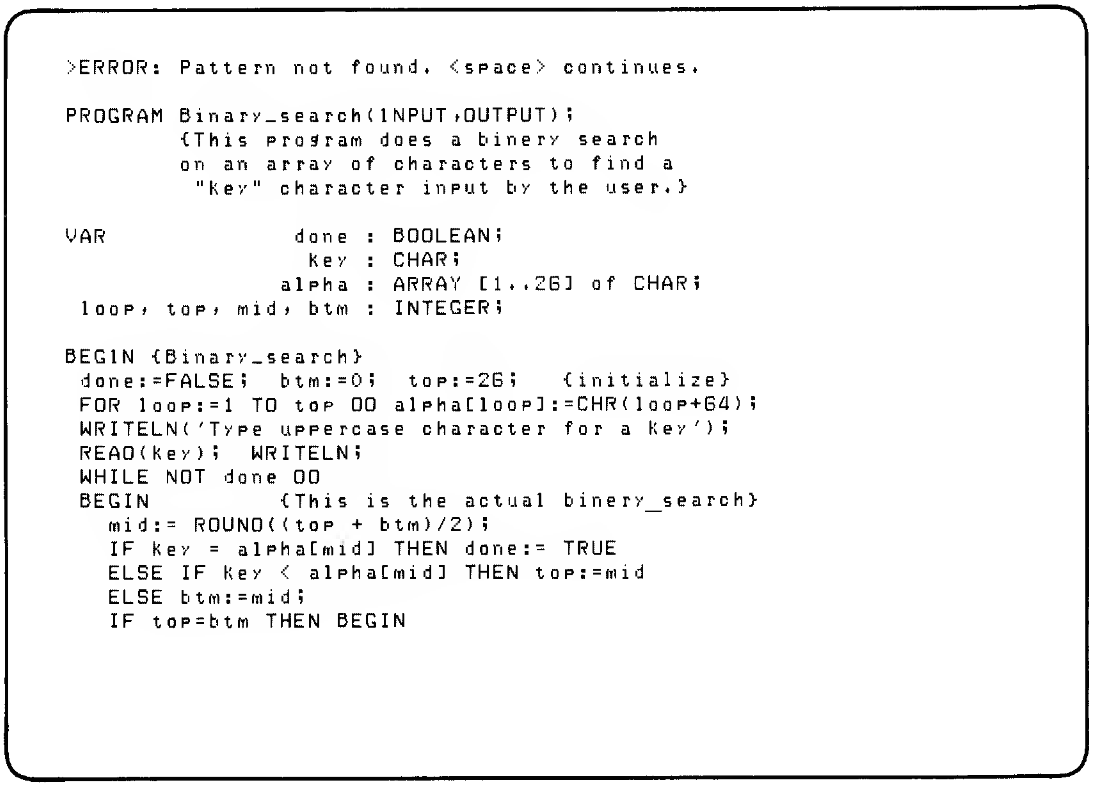
```
>Rpl[4]: <sh-sel> aborts,R replaces,' ' doesn't

PROGRAM Binary_search(INPUT,OUTPUT);
        {This program does a binery_search
        on an array of characters to find a
         "key" character input by the user.}

VAR              done : BOOLEAN;
                  key : CHAR;
                alpha : ARRAY [1..26] of CHAR;
 loop, top, mid, btm : INTEGER;

BEGIN {Binery_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN         {This is the actual binery search}
   mid:= ROUND((top + btm)/2);
   IF key = alpha[mid] THEN done:= TRUE
   ELSE IF key < alpha[mid] THEN top:=mid
   ELSE btm:=mid;
   IF top=btm THEN BEGIN
```

The first string inery has been replaced with inary, the cursor is now positioned behind the second occurrence of the target string and the prompt shows that you can make four more replacements. Press the space bar (represented by ' ' in the prompt) to leave the string unchanged and the screen now displays:



```
>Rpl[3]: <sh-sel> aborts,R replaces,' ' doesn't

PROGRAM Binary_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "Key" character input by the user.}

VAR             done : BOOLEAN;
                 Key : CHAR;
               alpha : ARRAY [1..26] of CHAR;
 loop, top, mid, btm : INTEGER;

BEGIN {Binery_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a Key');
 READ(Key);  WRITELN;
 WHILE NOT done DO
 BEGIN         {This is the actual binery search}
```

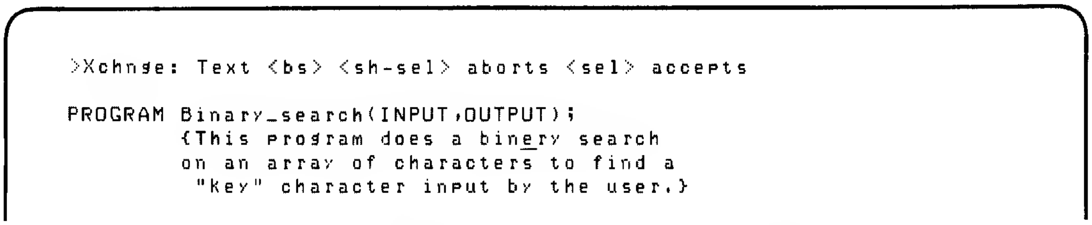
The cursor is now behind the occurrence of Binery following the BEGIN statement. Press R to replace this one and then press it again to replace the last occurrence of binery. The screen now displays:



```
>ERROR: Pattern not found. <space> continues.

PROGRAM Binary_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "key" character input by the user.}

VAR             done : BOOLEAN;
                 key : CHAR;
               alpha : ARRAY [1..26] of CHAR;
 loop, top, mid, btm : INTEGER;

BEGIN {Binary_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN         {This is the actual binery_search}
   mid:= ROUND((top + btm)/2);
   IF key = alpha[mid] THEN done:= TRUE
   ELSE IF key < alpha[mid] THEN top:=mid
   ELSE btm:=mid;
   IF top=btm THEN BEGIN
```

The prompt at the top of the screen tells you that the Editor could not find any more occurences of the specified string in the file. The cursor is positioned at the final occurrence of the string but it has not yet been changed. Press the space bar and the Editor prompt reappears, the final occurrence of the string gets replaced and the cursor remains at the same place on the screen.

To correct the spelling of binery (which was intentionally skipped), use the eXchange command. Move the cursor to the e in binery in the second line of your program. Now press X and the screen shows:

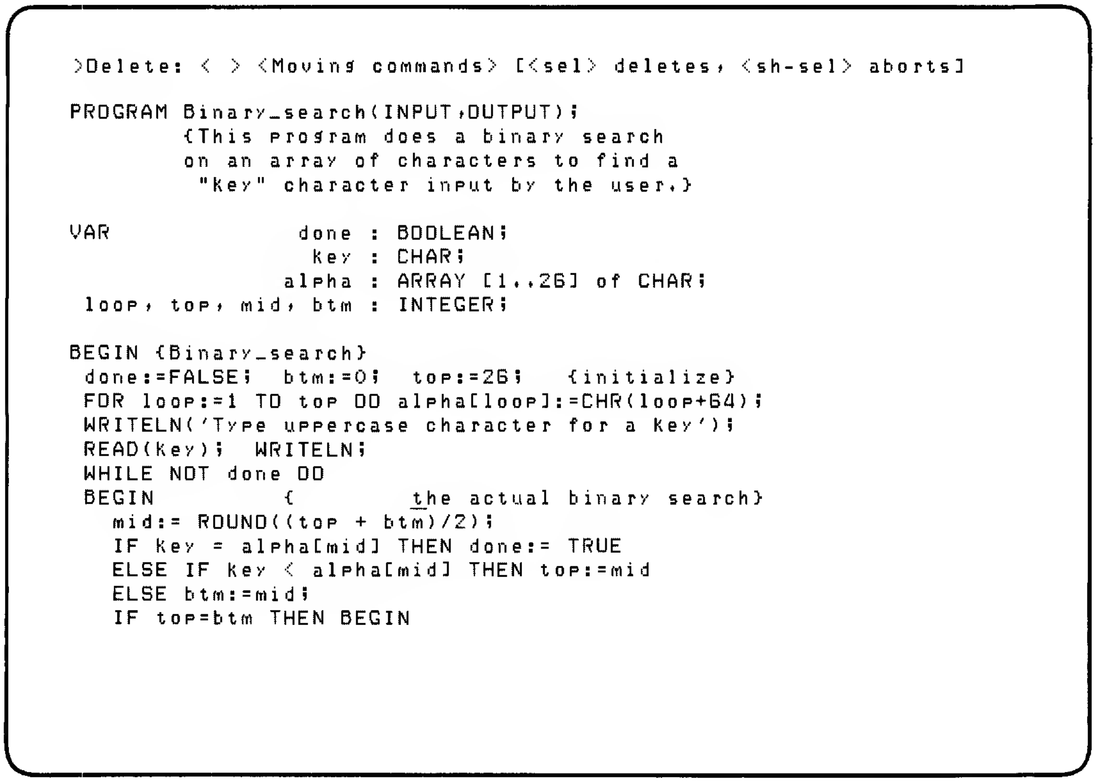


```
>Xchnge: Text <bs> <sh-sel> aborts <sel> accepts

PROGRAM Binary_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
         "key" character input by the user.}
```

Type the letter a and then press Select (EXECUTE). Pressing Select confirms changes made in eXchange and returns the Editor prompt. That's all there is to the eXchange command.

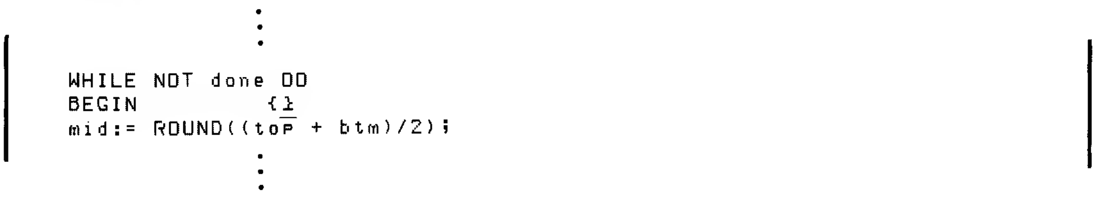
You should always position the cursor before initiating eXchange because this command cannot cross line boundaries; you can only make eXchanges on the line where the cursor is located.

The eXchange command is handy but the combination of the Insert and Delete commands is often a more effective way to change text. For example, to clarify the program by adding comments, position the cursor at the comment following the second BEGIN, press (D), and press (TAB) once. The screen displays:



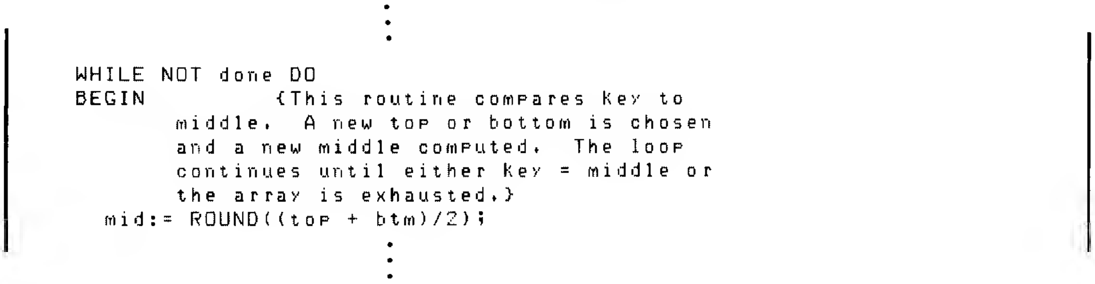
```
>Delete: < > <Moving commands> [<sel> deletes, <sh-sel> aborts]

PROGRAM Binary_search(INPUT,OUTPUT);
        {This program does a binary search
        on an array of characters to find a
         "key" character input by the user.}

VAR             done : BOOLEAN;
                 key : CHAR;
               alpha : ARRAY [1..26] of CHAR;
 loop, top, mid, btm : INTEGER;

BEGIN {Binary_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN          {        the actual binary search}
   mid:= ROUND((top + btm)/2);
   IF key = alpha[mid] THEN done:= TRUE
   ELSE IF key < alpha[mid] THEN top:=mid
   ELSE btm:=mid;
   IF top=btm THEN BEGIN
```

Using a combination of (TAB) and the space bar, delete everything between the two brackets and press (Select) ((EXECUTE)). Part of the screen looks like:



```
              .
              .
              .
WHILE NOT done DO
BEGIN         {}
mid:= ROUND((top + btm)/2);
              .
              .
              .
```

Press I to initiate the Insert command and notice how a space is opened between the brackets. Insertions always occur directly in front of the cursor's position when Insert is initiated. Now type in the text shown below and then press Select (EXECUTE) to complete the insertion.



```
                          .
                          .
                          .
     WHILE NOT done DO
     BEGIN          {This routine compares key to
            middle.  A new top or bottom is chosen
            and a new middle computed.  The loop
            continues until either key = middle or
            the array is exhausted.}
       mid:= ROUND((top + btm)/2);
                          .
                          .
                          .
```

Finding Patterns of Text

If you want to find a particular text pattern, you can use the Editor's Find command. This command is similar to the Replace command in that it begins looking for the pattern that you specify, beginning at the current cursor location; it also interprets the pattern you specify as a Token or a Literal, according to the current Token environment parameter setting.

Jump to the beginning of the file by pressing J and then B. Find the first occurrence of the word "loop" by pressing F and then typing these characters: `/loop/`. The /'s act as delimiters for the pattern that you want to find. These delimiters don't have to be the slash (/) character; the Find command uses the first character that you specify as the delimiter (except T while in Literal mode and L while in Token mode), so you will need to follow the pattern with the same delimiter. For example, you could have specified: `"loop"`.

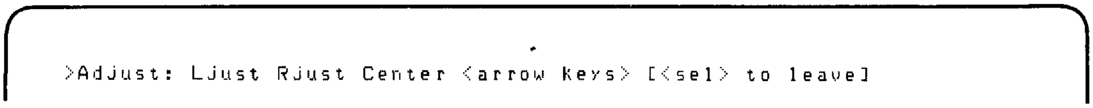
When the pattern is found, the cursor is placed at the beginning of the pattern. You can use commands to change the text (such as eXchange), or you can search for the next occurrence of the pattern by pressing F and then S (for Find Same).

If the pattern is not found, then you are prompted with the message: `>ERROR: Pattern not found. <space> continues.` Press space to answer the prompt, which puts you back into normal Edit mode.

Formatting Text

The Pascal Editor allows you to format text with the Adjust and Margin commands. Text is also formatted by inserting or deleting blanks where needed.

The Editor's Adjust command provides a means of shifting the starting column of a line of text left or right in the file. This command helps make your Pascal programs and other text more readable. To increase the clarity of our sample program, move the cursor to the word `mid` following the second `BEGIN` statement in the program. Press A and the Adjust prompt appears:



```
>Adjust: LJust RJust Center <arrow keys> [<sel> to leave]
```

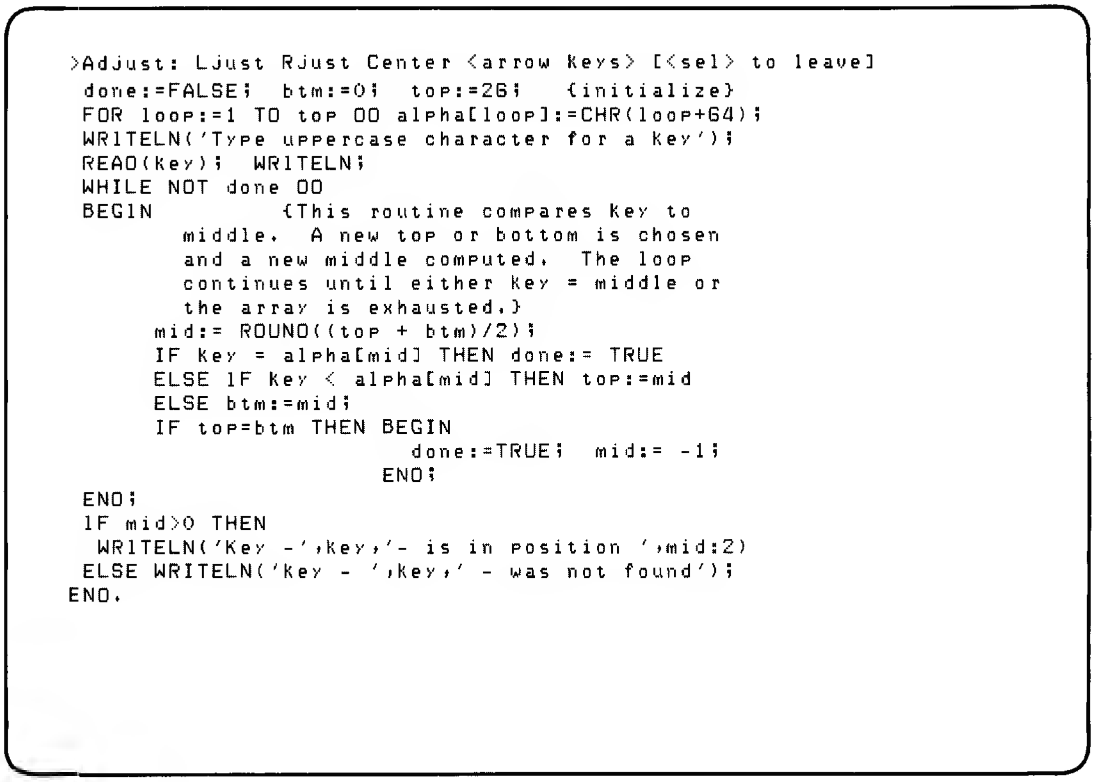
Experiment with the Adjust command by pressing L, R, or C. These options move text to the left, right, or center. The values used to shift the text are the Left and Right margins of the environment (discussed below). Any of the cursor arrow keys as well as BACK SPACE and the cursor wheel can be used to Adjust text. Now return the line to its original position and press Select. Repeat factors are available for use with the Adjust command so that many lines of text can be shifted at one time.

Note

Think twice before using Adjust with large repeat factors. This is because SHIFT-Select (SHIFT-EXECUTE), which usually aborts all changes made by a command, is not available for exiting the Adjust command. Therefore, to recover the original format of your text, you would have to Adjust it again.

Now that the line is in its original place, press A (to initiate Adjust), type 3 → (to indent the text three spaces to the right), and then type 6 and press ↓. Watch what happens: the cursor moves down six lines and shifts each line three spaces to the right. Thus, the Adjust command is useful for indenting entire blocks of text in a Pascal program.

The screen now looks like:



```
>Adjust: Ljust Rjust Center <arrow keys> [<sel> to leave]
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN         {This routine compares key to
        middle.  A new top or bottom is chosen
        and a new middle computed.  The loop
        continues until either key = middle or
        the array is exhausted.}
      mid:= ROUND((top + btm)/2);
      IF key = alpha[mid] THEN done:= TRUE
      ELSE IF key < alpha[mid] THEN top:=mid
      ELSE btm:=mid;
      IF top=btm THEN BEGIN
                        done:=TRUE;  mid:= -1;
                      END;
 END;
 IF mid>0 THEN
  WRITELN('Key -',key,'- is in position ',mid:2)
 ELSE WRITELN('Key - ',key,' - was not found');
END.
```

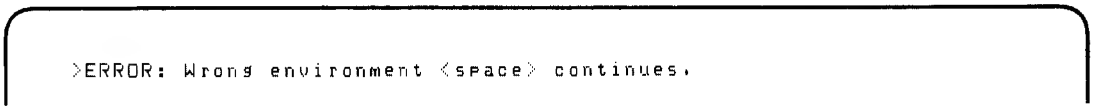
Press Select (EXECUTE) to terminate the Adjust command. If you wish to make adjustments in other parts of your text file, exit the Adjust command using Select before moving the cursor from one area to another. Otherwise you may make unwanted adjustments to your text.

The Margin command lets you margin and fill your text according to a predefined "environment". Margin operates on the paragraph where the cursor is located when M is pressed. A paragraph is any text delimited by any combination of blank lines, lines whose first non-blank character is the "command character" (see the Set environment command in the section "Editor Commands"), or the beginning or end of a file. The Margin command is executed completely by pressing M ; no parameters or options are available.

Entering the Editor without a workfile or a named file (as you did earlier in this session) automatically sets (or defaults) the environment to the "program" environment. This environment is optimized for writing programs. When the Editor is entered with either a file name or a workfile, the environment associated with that file is the current environment.

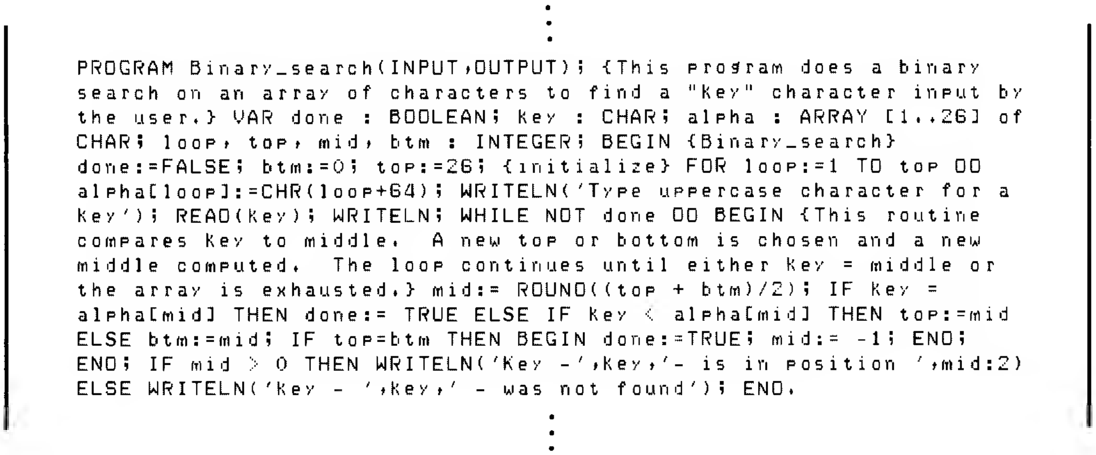
You can alter the environment at any time using the Set (Environment) command. Once you have altered or redefined the environment and saved a text file on a mass storage medium, that environment is stored along with the text file and is used whenever the Editor is entered with that file.

Since you entered the Editor without a file, your current environment is the Editor's "program" environment (the default supplied by the system). The Filling option of this environment is set to false (which disables the Margin command) so, if you press M , the screen displays:



```
>ERROR: Wrong environment <space> continues.
```

If Filling had been set True (with Auto-indent False), the Margin command would fill and Margin your program like this:



```
                                  .
                                  .
                                  .
PROGRAM Binary_search(INPUT,OUTPUT); {This program does a binary
search on an array of characters to find a "Key" character input by
the user.} VAR done : BOOLEAN; Key : CHAR; alpha : ARRAY [1..26] of
CHAR; loop, top, mid, btm : INTEGER; BEGIN {Binary_search}
done:=FALSE; btm:=0; top:=26; {initialize} FOR loop:=1 TO top DO
alpha[loop]:=CHR(loop+64); WRITELN('Type uppercase character for a
Key'); READ(Key); WRITELN; WHILE NOT done DO BEGIN {This routine
compares Key to middle.  A new top or bottom is chosen and a new
middle computed.  The loop continues until either Key = middle or
the array is exhausted.} mid:= ROUND((top + btm)/2); IF Key =
alpha[mid] THEN done:= TRUE ELSE IF Key < alpha[mid] THEN top:=mid
ELSE btm:=mid; IF top=btm THEN BEGIN done:=TRUE; mid:= -1; END;
END; IF mid > 0 THEN WRITELN('Key -',Key,'- is in position ',mid:2)
ELSE WRITELN('Key - ',Key,' - was not found'); END.
                                  .
                                  .
                                  .
```

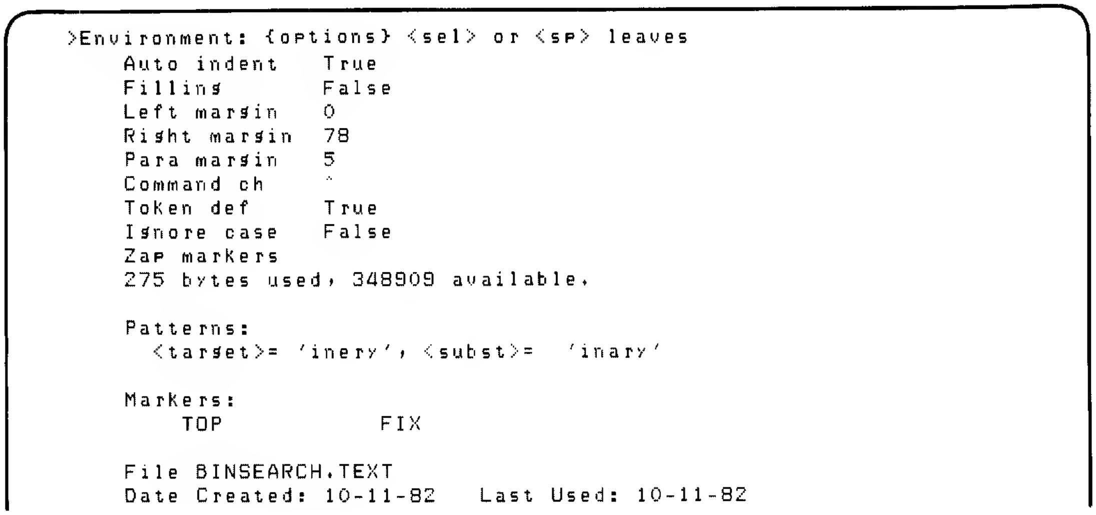
The previous display gives you some idea of how important it is to know what your environment settings are before using the Margin command. The only recovery from this operation is to use a combination of the Adjust and Insert commands to rebuild the text. If you have a copy of the original file available, you can exit the Editor without updating the file and reenter it with the old copy.

Note

The Insert command has effects similar to those of the Margin command when the Filling option of the environment is set to True and Auto-indent is False. Any time you do an Insert and confirm the operation by pressing (Select) ((EXECUTE)), both the inserted text and all the text that follows the insertion in that same paragraph are automatically margined.

The Margin takes place according to the Left and Right margin settings of the environment. If you begin an insertion and are not sure of the environment settings, press (SHIFT)-(Select) ((SHIFT)-(EXECUTE)) to exit the Insert command. This way, even if Filling is true, your text will not be margined. Then press (S) (E) to look at the environment settings.

When writing programs, your use of the environment and the Margin command will probably be more limited than when writing other kinds of text. To see how the program environment is configured, press (S) (E) (for Set Environment). The screen displays the default environment:

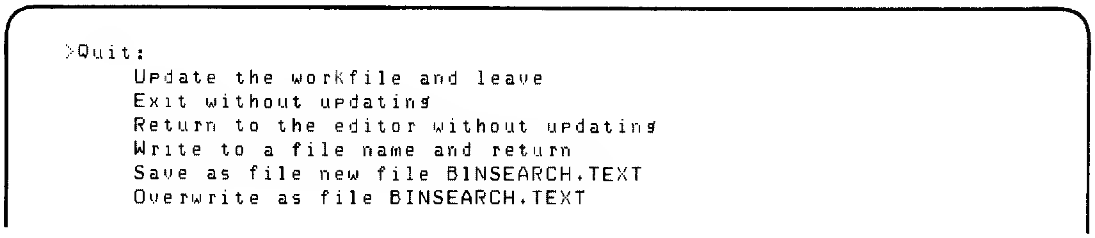


```
>Environment: {options} <sel> or <sp> leaves
    Auto indent   True
    Filling       False
    Left margin   0
    Right margin  78
    Para margin   5
    Command ch    ^
    Token def     True
    Ignore case   False
    Zap markers
    275 bytes used, 348909 available.

    Patterns:
      <target>= 'inery', <subst>=  'inary'

    Markers:
        TOP           FIX

    File BINSEARCH.TEXT
    Date Created: 10-11-82   Last Used: 10-11-82
```

Press the space bar to exit the environment display and the Editor prompt reappears along with your text. The cursor is at the position it was when the Set command was entered.

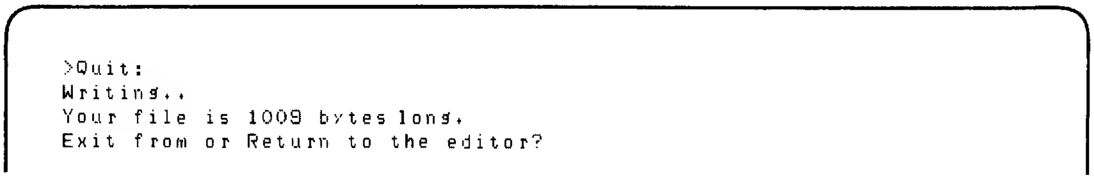
Exiting the Editor and Saving the File

Now that you have finished writing and editing the program, exit the Editor by pressing Q (for Quit). Make sure that the disc named DOC: is in the same disc drive you have been using. The screen displays:



```
>Quit:
     Update the workfile and leave
     Exit without updating
     Return to the editor without updating
     Write to a file name and return
     Save as file new file BINSEARCH.TEXT
     Overwrite as file BINSEARCH.TEXT
```

Respond by pressing S for Save. If you are on an SRM system, you would use the Overwrite option. The Overwrite option allows all duplicate links and passwords to a text file to remain accurate after a file has been modified. More information on these options is given in the command reference under the Quit command.



```
>Quit:
Writing..
Your file is 1009 bytes long.
Exit from or Return to the editor?
```

Your program text has been written to your disc and is accessible under the name BINSEARCH.TEXT on the volume DOC:.

If you are creating a file for use with another language system, such as BASIC or HPL, the file should be stored as an ASCII type file on a disc with a LIF directory. To accomplish that, use the Write option and name the file:

```
DOC:BINSEARCH.ASC
```

On a LIF directory, the Pascal system codes all the file names that end in a standard suffix. The coding scheme removes the period, appends the first character of the suffix to the file name, and pads the file name to ten characters with "_" (underscore characters). This allows you to specify file names up to 15 characters. They are encoded to the maximum ten characters for the LIF directory. The file system encodes the above file name to:

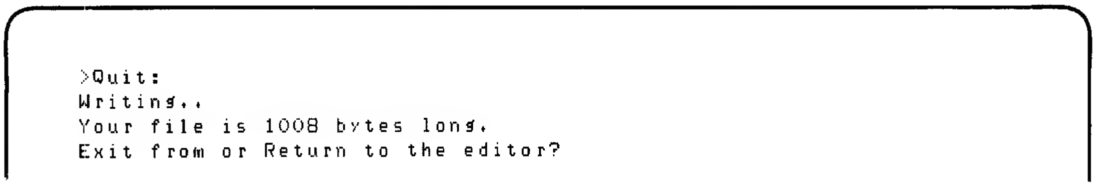
```
BINSEARCHA
```

In this case, the first character of the suffix is the tenth character so no "_" characters were added. This coding mechanism is invisible as long as you are using the Pascal system. When you catalogue your disc with other language systems, the coded version of the file name is observed.

Making a Backup Copy

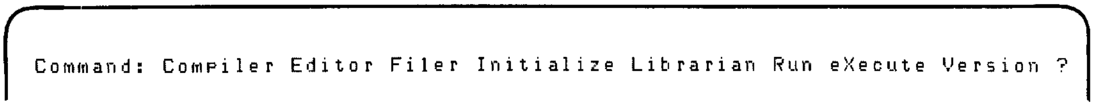
The most direct way to make a backup copy of your file is to press R (to return to the Editor) and then press Q (to initiate the Quit command). Each time you Quit the editor, you can make another copy of the file currently in the Editor.

Press W for the Write option, type in a name for your backup copy such as DOC:BINBKP and press Return or ENTER. If you have another disc handy, replace the DOC: volume with it, specify the name of the new volume along with a file name and press Return or ENTER. Remember the ten character limit for file names. After pressing Return or ENTER, the screen displays:



```
>Quit:
Writing..
Your file is 1008 bytes long.
Exit from or Return to the editor?
```

There are now two identical files on your disc(s) of the binary search program. Now press E (for the Exit option) and you will be returned to the Main Command Prompt:



```
Command: Compiler Editor Filer Initialize Librarian Run eXecute Version ?
```

All the Editor commands covered here are explained in further detail in the "Editor Commands" section. Less commonly used commands not presented in this sample session can also be found there.

A Closer Look

This section contains details about how the Editor works and includes information on the cursor, the screen, memory and file sizes and how the Editor allocates space for text files on a storage medium. The section also presents information on using workfiles in the Editor, on Stream files and on I/O errors that may occur when entering and exiting the Editor.

The Cursor

The cursor (the flashing underline symbol on the screen) is a reference point for all of the Editor's commands. The action associated with most commands occurs at the cursor position. Commands that perform actions on lines or paragraphs act upon the line or paragraph where the cursor is currently located.

You have complete control over the cursor through the arrow keys, the Tab and Return or ENTER keys, the space bar, the mouse, and the cursor wheel (also called the knob). The screen cursor's position determines where the Editor will act upon the text, thus reflecting the internal cursor's position (in the computer's memory).

The Anchor

You can also use the Zap command to delete text. With this command, all text between the current cursor position and the "anchor" is deleted. The anchor is set at the position of the latest Adjust, Find, Insert, or Replace command. (You can also find the position of the anchor by pressing =.)

If more that a line of text is to be deleted, you will be warned as follows: `>WARNING! Zap more than 80 chars? (y/n)` Press Y to confirm the Zap operation; press N (or space bar, etc.) to abort the Zap.

The Screen as a Window into a File

Text files are often too large to be shown all at once on the computer's screen (CRT), so the Editor uses the screen as a "window" which shows a portion of a file. Depending on which machine you have, your CRT can display lines of text that are either 49 or 79 characters long while in the Editor. If a line's length is greater than your display area, the Editor puts an exclamation point (!) in the last column to inform you that the entire line is not shown.

The screen is capable of displaying 25 lines of characters at a time. The Editor displays only 23 lines of text from a file since the top line is reserved for the system's prompt and the bottom line is reserved for the "type ahead" line. The type ahead line displays any characters input from the keyboard which have not yet been processed by the system. One other item is displayed in the lower right corner of the screen. This is a system status display or "runlight."

The screen generally displays the cursor and the text surrounding it. (The Set environment command is an exception to this). This means that you can move the window up and down through your text file by moving the cursor. Whether the text is on or off the screen, it resides in the computer's read/write memory and is easily accessed using either the cursor control keys or the various editing commands which reposition the window. When an Editor command operates on a portion of text it displays as much of that text as possible on the screen.

Memory and File Sizes

When the Editor is entered, the current text file is stored in the computer's read/write memory. All changes that occur to a text file (including text creation) take place in this memory and only become permanent when the Editor is exited and the contents of the text file are written from memory to a mass storage medium such as a flexible disc.

The maximum size of the text files that can be accessed or created by the Editor depend on the memory configuration of your system. This size can easily be determined using the Set (environment) command. The two environment headings, "bytes used" and "available", should be added together. The sum equals the maximum size (in bytes) of the text files which can be handled by the Editor.

If your text file approaches the maximum size while you are doing an insertion, the Editor displays the following message:

```
>ERROR: Finish the insertion  <space> continues.
```

This tells you that you are nearing the Editor's memory limits. If, after finishing the insertion, you attempt to initiate the Insert command again, the Editor informs you:

```
>ERROR:  No room to insert.  <space> continues.
```

Here is procedure to help you work around the Editor memory limits (whatever they may be on your machine):

1. Set a marker toward the end of your original file (to be used later).
2. Quit the Editor, Save the original file, and Exit the Editor completely (in order to reenter with a new file).
3. Reenter the Editor and press Return or ENTER (to create a new file).
4. Using the Copy from file command, specify your original file and your marker as follows: `FILENAME[MARK,]` and press Return or ENTER. The name of your file and marker will be different; this just shows you the general form. Notice that a second marker was not specified so that the copy takes place from the marker's location to the end of the original file.
5. Now, press J then E (to Jump to the end of your new file).
6. Press I (to initiate the Insert command).

Now you can continue inserting your text in your new file without too much loss of continuity. You may want to go back to your original file and delete the text that was copied into your new file so that it will not exist in both files.

Structure of Text Files

The Editor can read and write three types of files. The predominant file type is TEXT. The others are DATA and ASCII. TEXT files contain ASCII characters and are structured in a particular way.

In every text file, the first two blocks (or 1024 bytes) are reserved for information about the environment settings, the locations of markers in the file and other information the Editor needs to work with that file. Since the Editor allocates mass storage in two block increments, text files always contain an even number of blocks. Also, because the Editor reserves the first two blocks for file information, a file with only a single character will take up 4 blocks of storage space on a mass medium.

It is possible to create a text file that does not have `.TEXT` appended to the end of the file name. If, when exiting the Editor and specifying a file name using the `Write as ...` option, you place a period (.) at the end of the file name, the Editor will not append `.TEXT` to the file name. The file will appear to be a data file on the directory for the mass storage medium. (See the chapter on the Filer for more details on file types and the directory).

If you want to access this file with the Editor, you must specify the file name followed by a period when entering the Editor. If you do not use the trailing period, the Editor appends `.TEXT` to the name you type in and looks for a file with that name on the mass storage medium.

For example, suppose when exiting the Editor you answer the prompt for a file name with `DUX.` . Notice the period following the name. The Editor saves the file with the name `DUX` (it strips off the period) and does not append the `.TEXT` suffix. If you enter the Editor and want that file, you must specify `DUX.` . If you instead specify `DUX` (i.e., leave out the period), the Editor appends `.TEXT` to the name you typed and looks for a file with the name `DUX.TEXT`. It may even find a file with that name, but it will be a different file than the one saved by specifying `DUX.` .

ASCII files are structured differently. ASCII files on LIF discs are compatible with the BASIC and HPL language systems that run on your computer. ASCII files are created by writing to a file whose name ends in the suffix:

```
.ASC
```

When writing ASCII files, the Editor's environment information is lost.

Using Workfiles in the Editor

A workfile in the Editor is used as a "scratchpad" version of a text file. The workfile is useful because it is the default file in the Editor (as well as in many of the Pascal subsystems). Chapter 2 contains information about using workfiles in all the subsystems; only Editor-related workfiles are covered here.

There are two ways to enter the Editor: from the Main Command Level or from the Compiler subsystem (after the Compiler finds an error in the text file it is compiling). When entering from the Compiler, the text file being used is automatically read in. When entering the Editor from the main level, the system automatically looks for a workfile and, if it finds one, reads the contents of the file into the computer's memory. If the Editor does not find a workfile, it prompts you for a file name.

Exiting the Editor (using the Quit command) gives you the option of Updating the workfile. If the Editor was entered with a workfile (or if the Update option was used earlier in the same editing session), the Editor writes the contents of the text file in memory to the file called `*WORK.TEXT` on the system volume. When you are through with all your editing, it is a good idea to enter the Filer subsystem and Save the workfile.

Stream Files and the (ANY CHAR) Key

Stream files (covered in Chapter 1) can be created by the Editor to simulate a "batch" mode in which the computer executes the commands in the Stream file as if they were coming from the keyboard. The (ANY CHAR) key is useful in this regard. It can be used to generate characters which may not otherwise be attainable by regular keystrokes. For further information on the (ANY CHAR) key and Stream files, see Chapter 1.

I/O Errors (Entering and Exiting the Editor)

There are two general types of errors that can occur when entering the Editor. The first type of error is generated by the system when it cannot find the volume or file which you specified. The solution to this is to make sure that the proper volume is on-line. This type of error also occurs when a workfile exists but the Editor cannot find it because the medium containing that file is no longer on-line. When the Editor encounters this situation, it informs you that the workfile has been "lost" and then prompts you for a filename.

The second type of error possible while entering the Editor is a memory overflow condition. This happens only if the text file being read was created on a machine with more memory than the machine currently being used. Note that this condition is met if you use the Permanent command (at the Main Command Level — see Chapter 1) to load something into memory that was not there when you created the text file. Your machine now has less available memory so the space for text files is smaller.

When a memory overflow occurs while reading in the file, the Editor lets you continue the entry process even though the entire file has not been read into memory. However, upon exiting the Editor, the Save option is no longer available. This safeguard keeps you from accidentally overwriting your original file.

When exiting the Editor, a number of different errors are possible. If the Editor detects an error while writing the contents of the text in the computer's memory to a mass storage medium, it will display a self-explanatory error message.

Editor Commands

This section contains a brief overview and summary of all the Editor commands and a complete alphabetized description of the syntax and semantics of all the Pascal Editor commands and options.

Editor Command Summary

Text Modifying Commands

Copy – Insert text from the copy buffer or an external file in front of the current cursor location.

Delete – Remove text from the current cursor location to the location of the cursor when (Select) ((EXECUTE)) is pressed.

Insert – Inserts text in front of the current cursor location.

Replace – Replace the specified target string with the specified substitute string.

eXchange – Replace the text at the cursor with text typed from the keyboard, on a character-by-character basis.

Zap – Delete all text between the anchor and the current cursor location. (The anchor is set at the location of the latest Adjust, Find, Insert, or Replace command.)

Text Formatting Commands

Adjust – Adjust the column in which a line (or lines) start.

Margin – Format the paragraph the cursor is located to the margins in the current environment.

Miscellaneous Commands

Quit – Leave the Editor in an orderly manner. Provides various ways for saving the text currently in memory.

(STOP) ((SHIFT)-(CLR I/O)) – terminates the Editor subsystem, but the text is lost.

Set – Modify the environment or set markers in the text.

Verify – Update the displayed text to reflect the text stored in memory.

Cursor Keys

(TAB) – Move cursor to next tab position (fixed tabs) in the current direction.

(Return) or (ENTER) – Move cursor in current direction to the leftmost character in the next line.

Space Bar – Move cursor one character in the current direction.

Arrow Keys – Move cursor in the direction specified by the key.

Cursor Wheel – Moves the cursor like the arrow keys. Without (SHIFT), works like right and left arrows; with (SHIFT), works like the up and down arrows.

Cursor Positioning Commands

(=) – Typing (=) positions the cursor at the anchor. (The anchor is set at the location of the latest Adjust, Find, Insert, or Replace command.)

Find – Position the cursor after the specified target string.

Jump – Position the cursor at the beginning, the end, or the specified marker.

Page – Position the cursor ± 23 lines from the current location.

Command Syntax and Semantics

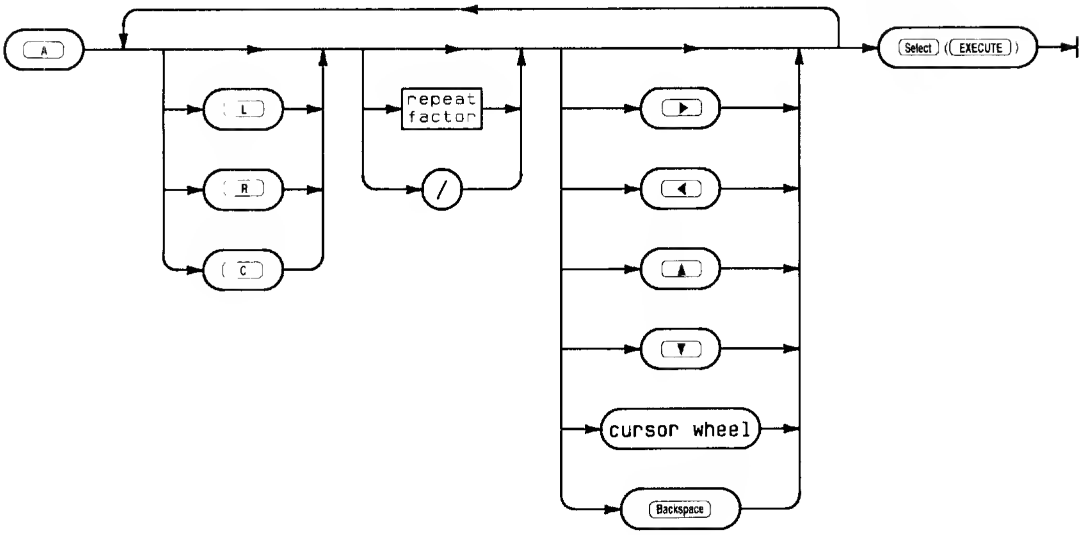
The Editor commands are presented in alphabetical order and described in a variety of formats to make them more useful to you. Each command's explanation includes: the command's name, a brief functional description, a diagram showing its legal syntax, the command's prompt (if any) and text which discusses using the command. Each of the command's options are also covered and some have examples to show the proper use of these options.

Alphabetical Listing of Editor Commands

Adjust
Copy
Delete
Equals (=)
Find
Insert
Jump
Margin
Page
Quit
Replace
Set
Verify
eXchange
Zap

ADJUST

Adjust horizontally shifts the starting column of one or more lines of text.



Item	Description/Default	Range Restrictions
repeat factor	integer numeric constant	1 thru 9999

Semantics

The Adjust prompt:

```
>Adjust: LJust RJust Center <arrow keys> [<exc> to leave]
```

The Adjust command provides a means of formatting text and enables you to make text more readable. Adjust uses the line position of the cursor when the command is entered as a starting point. A line-oriented command, Adjust lets you shift an entire line of text to the left or right using the →, ←, BACK SPACE, or cursor wheel. Repeat factors can be used with these keys to shift the text. For example, pressing 7 → results in the line of text shifting 7 spaces to the right.

Pressing A (for Adjust) and then L, R or C moves the line to the left margin, right margin or centers the line between the two margins. The margins used by these options are the Right and Left margins currently set in the environment (see Set command).

Typing a repeat factor and ↑ or ↓ causes that number of lines to be adjusted the same amount as the accumulated adjustments at that point. The slash (/) functions as an infinite repeat factor and can be used with ↑ and ↓. It causes adjustments to be made from the current line to either the beginning or the end of the text file, respectively. For example, pressing C / ↓ causes all the text between the current cursor position and the end of the file to be Centered according to the current margins.

Note

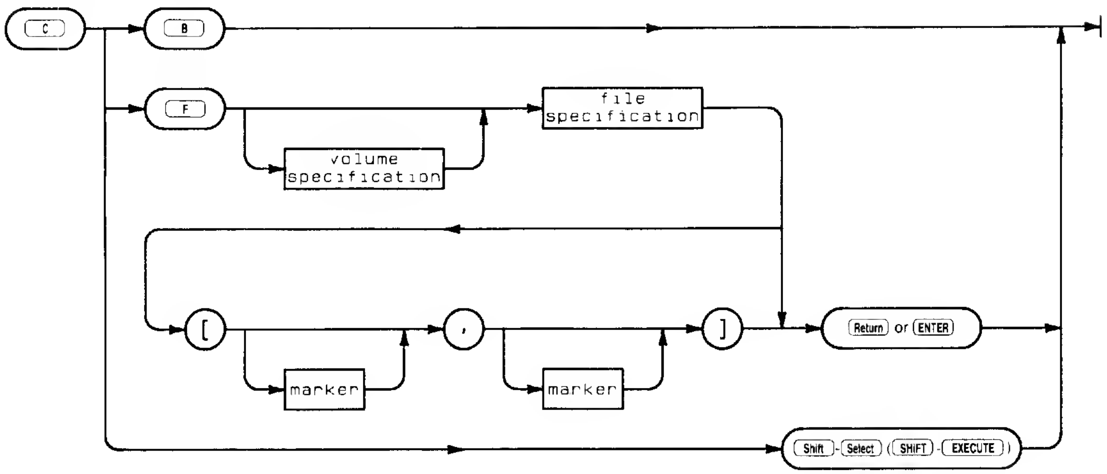
Take care when using large repeat factors or the slash (/) character when adjusting text. This is because the effects of the Adjust command cannot be aborted. Whatever adjustments are made become permanent unless the Adjust command is used again.

Adjust also sets the anchor used by the Zap command. Pressing = (the Equals command) moves the cursor to the position of the last Adjust unless the anchor has been reset by either a Find, Insert, or Replace command.

Leave the Adjust command by pressing Select (EXECUTE). The system stores the adjusted text in the computer's memory and the Editor prompt reappears.

COPY

Copy inserts text from a specified file or from the copy buffer.



Item	Description/Default	Range Restrictions
volume name	literal	any valid volume name.
file name	literal	any valid text file; do not enter .TEXT suffix.
marker	literal	1 to 8 ASCII characters excluding CHR (0) thru CHR (31) and CHR (127).

Semantics

The Copy prompt:

```
>Copy: Buffer File <sh-sel>
```

The Copy command provides a way of moving or duplicating text in a file and copying text from another file. These are the Buffer and File options. Pressing C (for Copy) and B (for Buffer) results in the contents of the copy buffer being written to your current text at the cursor position when the command was entered. The screen displays the new text and the Editor prompt.

The copy buffer is filled with the text involved in the most recent Delete, Insert or Zap command and its contents are cleared with the Margin command. Margin clears the copy buffer regardless of the settings in the environment. Doing a Copy (from a File) also clears the copy buffer. A subsequent Copy from Buffer command generates the message:

```
>ERROR: Invalid copy. <space> continues.
```

Any subsequent Delete, Insert or Zap refills the buffer (destroying its previous contents) and copying from a file clears the contents of the buffer. However, doing a Copy (from Buffer) does not alter the buffer's contents. Neither do any cursor control movements or commands. Therefore, you can make multiple copies of the same text in different locations by repeatedly positioning the cursor and pressing C B.

To Copy from a file, press C F. The screen displays:

```
>Copy: File [marker,marker] ?
```

The Editor is requesting a volume name, file name, and two marker names. The volume name may be omitted if the file in question is on the default volume. The volume (specified or default) must be on-line. Specification of the two previously set markers (see Set command) is optional but, if given, the marker names must be enclosed in square brackets and separated by a comma. Remember, only TEXT type files have markers.

If markers are specified, only the text between those two markers is copied. If no markers are specified, the entire file is copied. Only one marker has to be specified. If it is the first marker (i.e., followed by a comma), the text is copied from the marker position to the end of the specified file. If only the second marker is given (i.e., preceded by a comma), the text is copied from the beginning of the specifed file to the position of the marker. The copy occurs at the cursor's position when the Copy command was entered. You can exit the command before all specifications are complete by pressing SHIFT-Select.

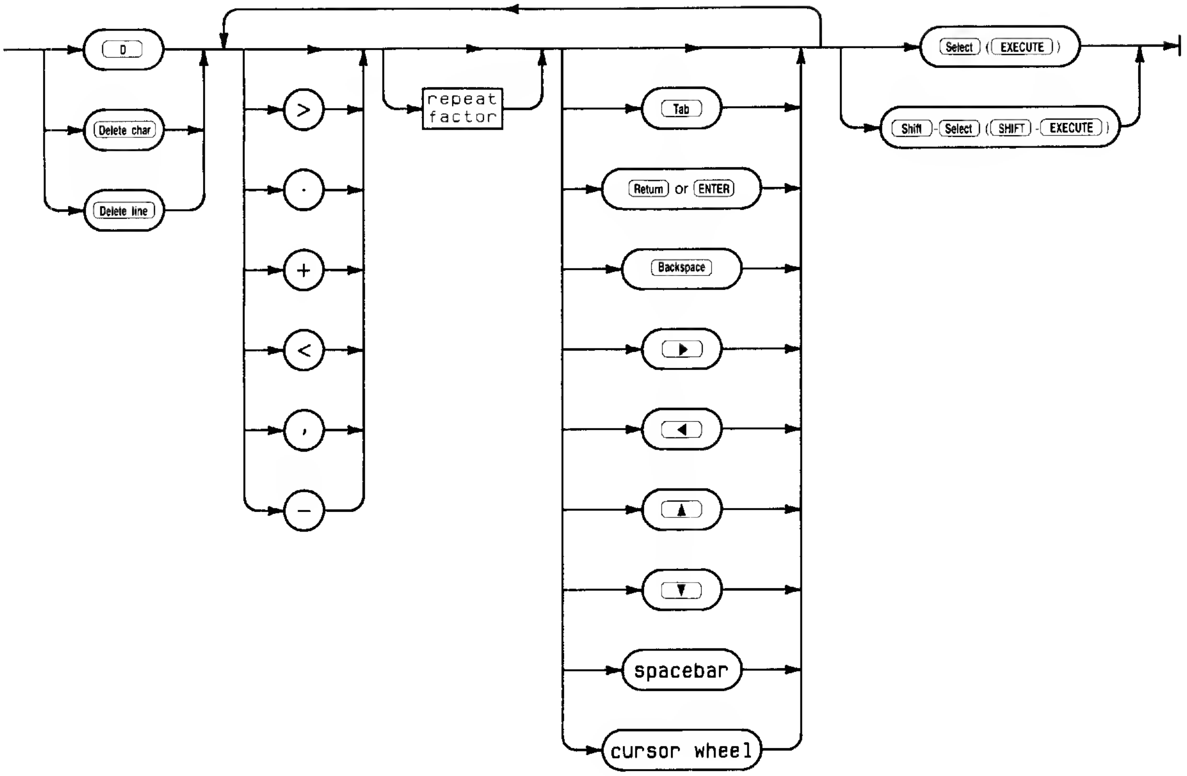
After typing the appropriate information and pressing Return or ENTER, the Editor displays:

```
>Copy...
```

This shows that the specified text is being copied into your current text. When the operation is complete, the Editor prompt reappears and the screen displays all or part of the text that was copied.

DELETE

Delete removes text from the current file.



Item	Description/Default	Range Restrictions
repeat factor	integer numeric constant	1 thru 9999 (1 thru 4095 for TAB)

Semantics

The Delete prompt:

```
>Delete: < > <Moving commands> [<exc> deletes, <sh-exc> aborts]
```

The Delete command enables you to remove text and fills the copy buffer with the deleted text. Delete uses the cursor position when the command is entered as a starting point. Subsequent cursor movement by any cursor control key causes text to be removed between this point and the new cursor position. Text can be recovered by moving the cursor back toward the starting point.

Direction applies in the Delete command and is shown by (>) (forward) or (<) (backward) in the Delete prompt. If forward, movement occurs from the cursor toward the end of the file; if backwards, movement is from the cursor toward the beginning of the file. Movement generated with the TAB, Return or ENTER, and space bar takes place in the direction shown. Direction can be changed while in the Delete command by pressing > , or + (for forward) or < , or - (for backward).

Repeat factors are available within the Delete command. For example, pressing D (for Delete) and then 9 Return ENTER will remove 9 lines of text in the current direction starting at the cursor position.

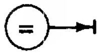
Delete fills the copy buffer with the deleted text and thus provides a means of moving or duplicating text. See the example in the section "A Sample Editor Session".

To exit the Delete command press Select (EXECUTE) or SHIFT-Select (SHIFT-EXECUTE). Select confirms the deletion, returns the Editor prompt and displays the cursor at its position when Select was pressed. SHIFT-Select aborts all changes made since Delete was entered, returns the Editor prompt and displays the cursor at its position when Delete was entered.

Note that the copy buffer is filled by whatever is deleted; whether the command is exited with a Select or SHIFT-Select makes no difference to the copy buffer.

EQUALS (=)

EQUALS positions the cursor at the anchor's location.

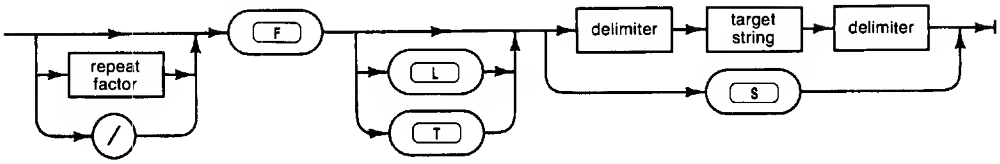


Semantics

The equals sign (=) is a cursor positioning command. It moves the cursor to the beginning of the most recent item Adjusted, Found, Inserted, or Replaced. Pressing = causes the cursor to jump to the location of this ''anchor'' and the Editor's prompt is displayed. This is the anchor used by the Zap command.

FIND

Find moves the cursor to an occurrence of a specified string.



Item	Description/Default	Range Restrictions
repeat factor	integer numeric constant	1 thru 9999
delimiter	literal (see glossary)	any valid delimiter; must be used in matched pairs.
target string	literal	1 thru 128 characters

Semantics

The Find prompt:

```
    >Find[1]: L <target>=>
or  >Find[1]: T <target>=>
```

The prompt displayed depends on whether the "Token" definition in the Editor's environment is set to true or false. If set to true, the first prompt is displayed; if false, the second is shown. These are explained below.

In its simplest form, the Find command is executed by pressing F and specifying a string surrounded by delimiters. Upon typing the final delimiter, the cursor is positioned at the end of the first occurrence of the specified string in the current direction.

The Find command moves the cursor and sets the anchor (used by Zap) at the location of the target string. In this context, a "string" is a contiguous series of non-control ASCII characters surrounded by delimiters. Delimiters are separators that signify to the Editor the beginning and end of the string such as the characters / ' , and > (more are listed in the glossary).

Avoid using a delimiter that appears in your string. Delimiters must be matched pairs; if you use $ to signify the beginning of a string, you must use $ to signify the end of the string. The maximum length of a target string is 128 characters.

The Find prompt shows the current direction. When searching for a string occurence, Find looks for that string between the cursor position when the command was entered and either the end of the file (if direction is forward >) or the beginning of the file (if direction is backward <).

Repeat factors are available with the Find command but must be typed before the Find command is initiated. If a repeat factor is used, the Editor positions the cursor at the end of that occurrence of the string. For example, typing 8 F /the/ results in the cursor being positioned at the end of the eighth occurrence of the. The slash (/) operates in a similar way but signifies the last occurrence of the specified string in the current direction. If no repeat factor or slash character is specified it defaults to the value 1 and the Editor attempts to find the first string occurrence. The Find prompt displays this value in square brackets.

After pressing F, the prompt on your screen contains either an L or T for "literal" or "token" modes. Literal and token are interdependent; if one option is shown as available, the other is automatically the default. If L is shown in the prompt and you want to use the token search mode, simply type in the target string surrounded by delimiters. The search will take place in the default mode (in this case, token). To do the same search in the literal mode, press L then type in the string as before. The Find command then searches for a literal form of the string.

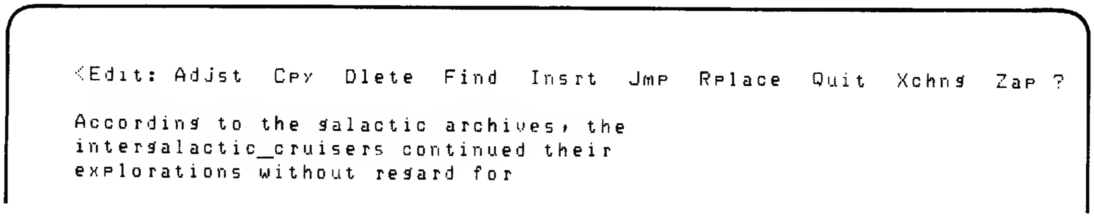
A literal string is exactly that — a literal string of characters either isolated or imbedded in a word or paragraph. **A token string** is one which is isolated by delimiters. Delimiters in this context are any ASCII characters except numbers or alphabetic characters. Blanks are common delimiters in English text because they separate words.

To illustrate literal and token searches, the following example assumes the direction is forward (>) with the cursor located at or before the start of the sentence shown. In the sentence That's my hat!, a token search for hat moves the cursor behind the last word hat in the sentence whereas a literal search for hat moves the cursor behind the hat imbedded in That's.

The "same" option is another feature of the Find Command. Same refers to the most recent target string used in either the Find or Replace command. Suppose you typed the sequence F L *galactic*. After pressing the final delimiter (*), the Editor moves the cursor behind the first literal occurrence of the target string galactic. Then typing F L S results in the cursor moving behind the next literal occurrence of the same target.

When using the "same" option with the direction set backward (<), use a repeat factor of two before initiating the Find. Otherwise, the Editor finds the previous occurrence since it searches between the cursor's current position and the beginning of the file.

Suppose after the first Find your screen displays:



```
<Edit: AdJst  Cpy  Dlete  Find  Insrt  Jmp  Rplace  Quit  Xchng  Zap ?

According to the galactic archives, the
intergalactic_cruisers continued their
explorations without regard for
```

If you press F L S to Find the same literal string, the cursor position on the screen does not change. To find the next occurrence, type 2 then press F L S. Since the direction is backwards and the Find command always positions the cursor behind the target string, using a repeat factor of 2 skips to the next effective occurrence of the literal `galactic`. Repeated searches in the forward (>) direction operate in a straightforward manner.

Note

If a Replace has been done since the last Find operation, the target string used by the "same" option is now the target specified in the Replace command.

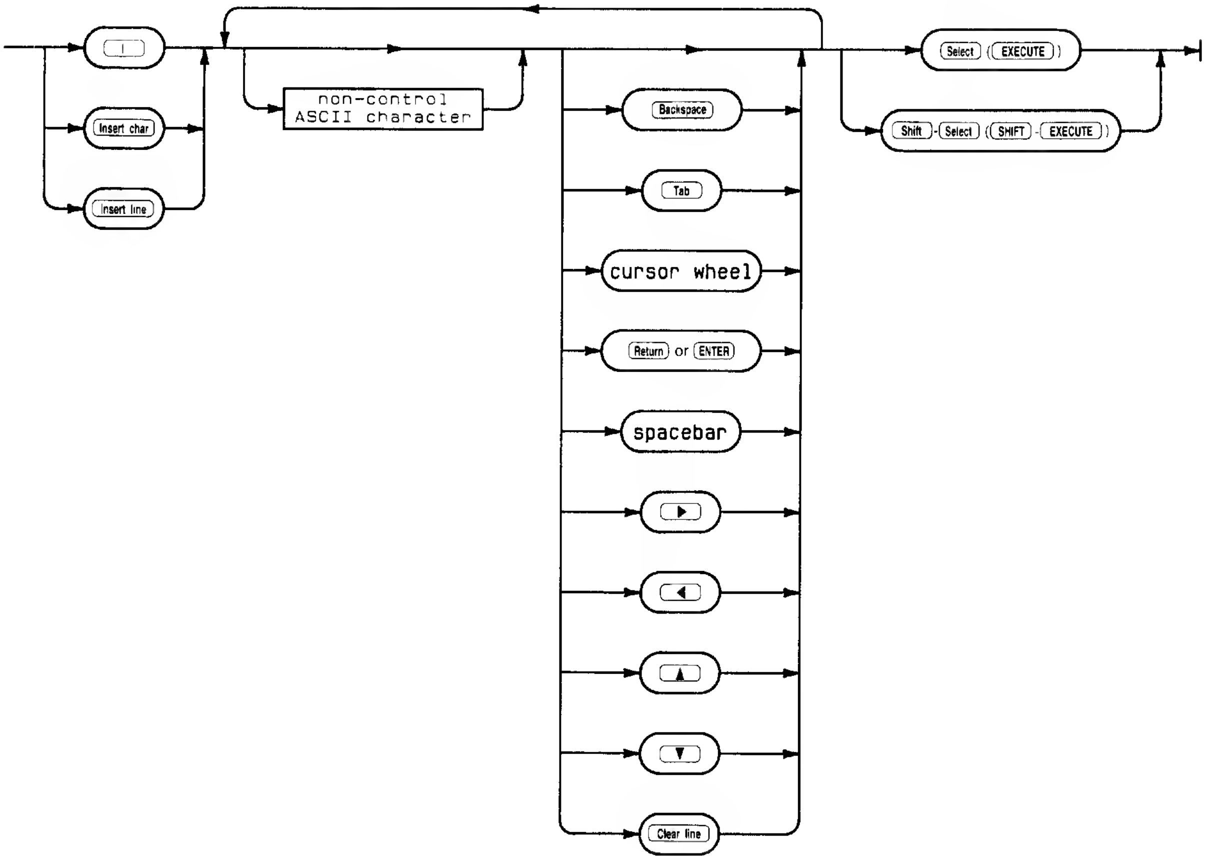
Searches are sensitive to the case of the characters (upper and lower case) unless Ignore case and Token are set to True in the Environment. Type S and E to Set the Environment. Type I and T to set Ignore case to True. Type T and T if Token is not already True. After these two conditions are met, the Editor treats both the target string and each token string as uppercase.

Find is one of the commands that sets the anchor used in the Zap command and accessed by the Equals command.

SHIFT-Select (SHIFT-EXECUTE) can be used to abort the Find command before all specifications are complete. If SHIFT-Select is used, the target pattern used by the "same" option remains unchanged. Select cannot be used with the Find command. The command is executed immediately when the final delimiter (or S if "same" is used) is typed.

INSERT

Insert opens a window in the current file for the subsequent insertion of text.



Item	Description/Default	Range Restrictions
non-control ASCII character	literal	Any valid ASCII character excluding: CHR (0) thru CHR (31) and CHR (127).

Semantics

The Insert prompt:

```
>Insert: Text <bs>, <clr ln> [<sel> accepts, <sh-sel> escapes]
```

The Insert command opens a window in the text file directly in front of the cursor position for text creation. When initiated (by pressing I or INS CHR or INS LN), the text from the cursor to the right edge of the screen is shifted to the right. Insertion always takes place directly in front of the cursor location when Insert was entered. Any sequence of non-control ASCII characters may be inserted and any cursor control key may be used. However, unless the movement generated by the cursor control keys is backward, they produce question marks (?) in the text. You can BACK SPACE to delete a character or press CLR LN to delete the most recently inserted line. CLR LN is available only after a line of text has been inserted. Backspacing past the point at which Insert was entered is not possible.

The way in which text insertion takes place depends on flags or parameters set in the Editor's environment. These flags have default values supplied by the Editor but can be changed with the Set command. The ones that concern you here are Filling and Auto indent. These two options generally have opposite values. Most of what you need to know about Filling and Auto indent can be summed up in one sentence: If you are writing program source text, set Filling to false and Auto indent to true; if writing regular text, set Filling to true and Auto indent to false.

Filling, when set true, performs both "wrap around" and "margining" functions. As inserted text approaches the Right margin (another environment option), the Editor attempts to fit the words on the current line. If a word would cause the line to extend beyond the right margin, it is automatically shifted to the next line (i.e., the system supplies a carriage return and a line feed). When the insertion is completed by pressing Select (EXECUTE), all text following the cursor in the current paragraph is margined. Margining adjusts the text to fit between the environment's margins and also compresses blanks in the text. You can have two blanks following the four characters: ? , : !. All other blanks are compressed into a single blank character.

Note that the Editor's definition of a paragraph is **ANY** text delimited by any combination of blank lines, lines having the Command character as the first non-blank character in a line, or the beginning and end of a text file. The Command character is yet another of the environment's options; see the Set command for more details.

Note

As the definition of a paragraph infers, the Editor does not distinguish tables from other kinds of text material. Any insertions within a table will result in the table being margined (i.e., collapsed) if Filling is set to true and the insertion is exited with Select (EXECUTE). Use the Set command to set Filling to False before inserting in a table or list. (SHIFT-Select will **NOT** restore the text to its original state after a paragraph has been margined).

If Filling is false, a beep is generated as you approach the end of the line, signaling you to press Return or ENTER the same way a bell on a typewriter does. If you continue to insert text past the last visible column on your screen, the Editor accepts the characters and shows you that they are outside of the display area by placing an exclamation point (!) in the final column. To access these characters, complete the insertion by pressing Select (EXECUTE), position the cursor before the last word on the line and press I followed by Return or ENTER to insert a carriage return.

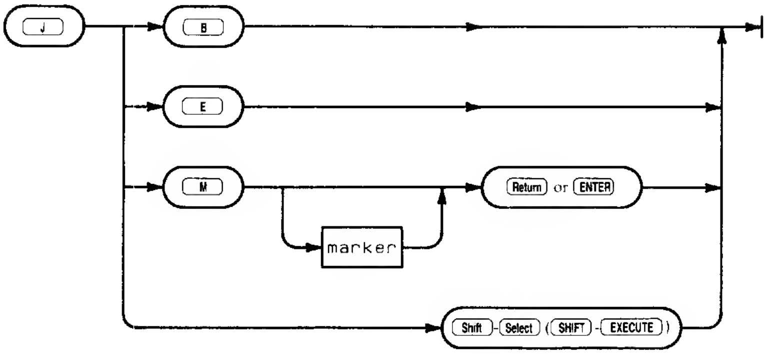
If Auto indent is true, pressing (Return) or (ENTER) automatically places the cursor in the same starting column as the previous line. When Auto indent is false, the cursor is positioned according to either the Left or Paragraph margin in the environment.

If Insert is exited with (Select) ((EXECUTE)) (and Filling is true and Auto indent false), all text following the insertion in the same paragraph is margined according to the Right, Left and Paragraph margin values in the environment. Also, the entire insertion is stored in the copy buffer so you can copy the same text elsewhere if you wish. If Insert is exited with (SHIFT)-(Select) (regardless of the options set), all changes are aborted and the text and cursor appear as they did when the command was entered.

The Insert command sets the anchor (used by the Zap command) at the position where Insert was initiated. The anchor is set regardless of whether (Select) or (SHIFT)-(Select) is used to exit the command.

JUMP

The Jump command repositions the cursor.



Item	Description	Range Restrictions
marker	literal	1 to 8 ASCII characters excluding: CHR (0) thru CHR (31) and CHR (127).

Semantics

The Jump prompt:

```
>JUMP: Begin End Marker <sh-sel>
```

The Jump command moves the cursor to the beginning or end of a text file or to a previously defined marker. The command has no other effects; it merely repositions the cursor. To Jump to the beginning of your file, press J B. To Jump to the end of your file, press J E.

You can also Jump to a marker by pressing J M and typing the name of any previously set marker in the file followed by Return or ENTER. Marker names are defined with the Set command. A legal marker name is any sequence of up to eight non-control ASCII characters (control characters are deleted by the system). They can actually be longer than this but the Editor only pays attention to the first 8 and truncates the rest.

Also, marker names are not case sensitive. The Editor converts all marker names to uppercase letters so they can be typed using any desired combination of uppercase and lowercase letters. There is a 10 marker limit per text file. See the Set command for more information on markers.

MARGIN

Margin formats all text in the current paragraph to fit the margins set in the environment.

Semantics

Margin is disabled and the system generates an error message unless the environment's Auto indent is false and Filling is true when the command is executed.

The Margin command provides a means of formatting paragraphs in your file. A paragraph is defined by the Editor to be **ANY** text delimited by any combination of blank lines, lines having the Command character as the first non-blank character in a line, or the beginning and end of a text file. See the Set command for details on the Command character.

Upon initiating Margin (by pressing M), the Editor takes all the text in the current paragraph (the one where the cursor is) and forces it to fit within the Left, Right and Paragraph margin boundaries of the environment. After margining, the first line of the paragraph begins at the column specified by the Paragraph margin setting and the rest of the text conforms to the Left and Right margin settings. If a word would exceed the Right margin it is "wrapped around" to the next line.

Two blanks are allowed following the four characters: ? . : !. All other blanks are compressed into a single blank character.

Since the Command character in the environment delimits a paragraph, you may want to use it as the first character in each line of tables or lists which you do not want margined. See the Set command for more information on the Command character.

Note

If a table or list fits the definition of a paragraph, the Margin command will definitely margin that text. Exiting the Insert command with Select (EXECUTE) also uses some of the Margin routine so be aware that these commands can potentially "collapse" a table or list.

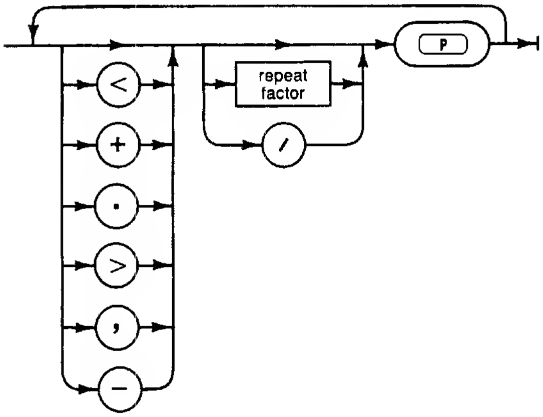
The Margin command has no parameters and its effects cannot be aborted. When writing program text or tables, it is advised that Auto indent be set true, Filling be set false and the Paragraph margin be equal to the Left margin.

Note

The Margin command clears the contents of the copy buffer regardless of the settings of the Auto indent and Filling options.

PAGE

Page moves the cursor one or more pages (23 lines) in the current direction.



Item	Description/Default	Range Restrictions
repeat factor	integer numeric constant	1 thru 1000

Semantics

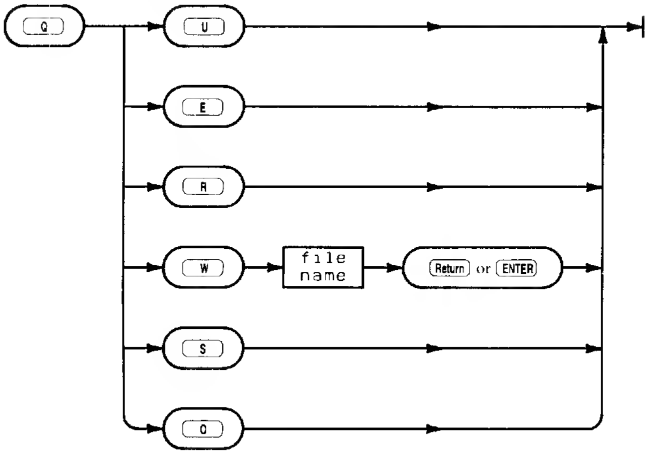
The Page command lets you move rapidly through a text file by repositioning the cursor one or more pages (23 lines of text) forward (>) or backward (<) in a file. Page is executed by pressing P and its movement occurs relative to the position of the cursor. Page moves the cursor in the direction displayed by the Editor prompt when the command is entered. The direction can be changed by pressing > . or + for forward or < , or - for backward.

Repeat factors are available in the Page command. For example, to move the cursor 3 "screens" or pages in the file, press 3 P. The slash character (/) can be used in place of an integer repeat factor. Pressing / P results in the cursor moving to the end of the file (if direction is >) or the beginning of the file (if direction is <). If neither repeat factor nor slash is specified, the default is 1 and the cursor moves one page.

Select (EXECUTE) and SHIFT-Select (SHIFT-EXECUTE) are not available in the Page command. The command is immediately executed when P is pressed.

QUIT

QUIT leaves the Editor with various exit options.



Semantics

```
>Quit:
     Update the workfile and leave
     Exit without updating
     Return to the editor without updating
     Write to a file name and return
     Save as file new file BINSEARCH.TEXT
     Overwrite as file BINSEARCH.TEXT
```

The Quit command allows you to exit the Editor and store your file on a mass storage medium. The last two quit options shown above are only available if the file existed before the editing session.

Quit is initiated by pressing Q from the Editor's prompt. Choose any of the options displayed by pressing the first letter of the option.

Pressing U (for **Update**) results in the contents of the text in the computer's memory being written to a text file on the system volume under the name `WORK.TEXT`. This workfile may or may not be associated with another file name (see the Get and Save commands in the Filer chapter). After writing the file, the system reports the file's size (in number of bytes and blocks) and displays the main command prompt.

Pressing E (for **Exit**) either immediately exits to the main level or displays:

```
Are you sure you want to exit without updating?
  Type Yes  to Exit without update
  Type No   to Return to Editor
```

This message is displayed only if changes have been made to the text file in the current editing session. If no changes have been made, the system immediately goes to the main level when E is pressed. It also exits to the main level if you respond by pressing Y. Responding with N returns you to the Editor.

Pressing R (for **Return**) returns you to Edit mode with the cursor located where it was when Quit was entered.

Pressing W (for **Write**) causes the system to prompt you for a file name. A complete file name is needed. If a volume ID is not given, the default volume is used. The volume PRINTER: may be specified. This results in the file being listed to the printer.

If you use the Write option and the file already exists, the Editor displays this prompt:

```
>Quit:
FILE.TEXT exists ...
  Rewrite then purge old
  Overwrite
  Purge old then rewrite
  None of the above
```

Rewrite then purge old is like the Save command. An attempt is made to write the new file before purging the old.

Overwrite removes the original file and then attempts to write the new version in its place. On SRM units, duplicate links and passwords will be preserved. On a disc, the file may not fit if the new version is larger than the old.

Purge old then rewrite removes the original file and then attemps to write the new file in the biggest space on the disc. This alternative gives you the best chance that there will be room for the new file.

Whether you "Overwrite" or "Purge old then rewrite", the original copy of the file is gone and the only copy of the file is in the Editor's memory. It is advisable to save it on another disc as soon as possible.

None of the above returns you to the Editor. You may Quit again and write the file with a different name.

Pressing S (for **Save**) results in the file being written to the original volume and file.

If you try to Save a file and you get the message:

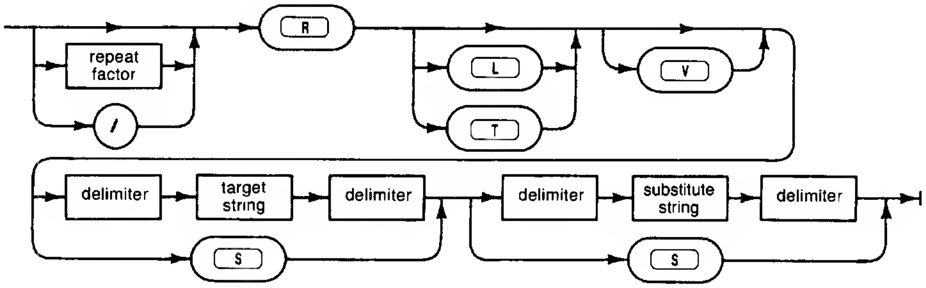
```
>ERROR: No room on vol <space> continues.
```

Press the spacebar to continue. You could put in another disc with enough space, then Quit and Save it on the new disc. Alternatively, you can Quit and Overwrite the file.

Pressing O (for **Overwrite**) is designed for SRM systems. The Overwrite option allows all duplicate links and passwords to remain accurate. On a disc, Overwrite may not work if the file has been enlarged. If this happens, press the spacebar to continue, Quit and Save again. The previous Overwrite removed the original file. Now the Save will try to save the file in the largest space on the disc. If this does not work, you must put the file on another disc.

REPLACE

Replace does one or more substitutions of a specified string for another string.



Item	Description/Default	Range Restrictions
repeat factor	integer numeric constant	1 thru 9999
delimiter	literal (see glossary)	any valid delimiter; must be used in matched pairs.
target string	literal	1 thru 128 characters
substitute string	literal	1 thru 128 characters

Semantics

The Replace prompt:

```
   >Repl[1]: L  V  <targ><sub>=>
or >Repl[1]: T  V  <targ><sub>=>
```

The prompt displayed depends on whether the "Token" definition in the Editor's environment is set to true or false. If true, the first prompt is displayed; if false, the second is shown. These are explained below.

The Replace command allows you to substitute one string for another in your text file. The anchor (used by the Zap command and accessed by the Equals command) is set at the location of the replacement. Replacements can be done to a single, all, or only certain occurrences of a string.

In its simplest form, the Replace command is executed by pressing R and specifying two strings — a target and substitute — each surrounded by delimiters. The target and substitute strings may be different sizes. Upon typing the final delimiter, the first occurrence of the target string is replaced by the substitute string and the cursor is positioned at the end of the substitution.

A target string (the one that you want replaced) must be supplied. A string is a contiguous series of non-control ASCII characters surrounded by delimiters. Delimiters signify the beginning and end of a string and are characters such as: / ' , and > (more are listed in the glossary).

A substitute string (what you want the target string changed to) must also be supplied with delimiters. The substitute may be an empty (null) string.

Avoid using a delimiter that appears within your string. Delimiters must be matched pairs, i.e., if you use $ to signify the beginning of a string, you must use $ to signify its end. The substitute string can have a different set of delimiters than the target string and the two strings may be of different sizes. The maximum length of either string is 128 characters.

After pressing R, the prompt on your screen contains either an L or T for "literal" or "token" modes. Literal and token are interdependent; if one option is shown as available, the other is automatically the default. If L is shown in the prompt and you want to use the token search mode, then type in the two strings and their delimiters. The replacement takes place in the default mode, in this case, token. To do the same replacement in the literal mode, press L and type in both strings as before. The Replace command then searches for a literal form of the string.

A literal string is exactly that — a literal string of characters either isolated or imbedded in a word. **A token string** is usually a word (a string isolated by delimiters). Delimiters, in this context, are any ASCII characters except numbers or alphabetic characters — they do not have to be matched pairs. Blanks are the most common delimiters in English text because they separate words.

To illustrate literal and token replacements, the following example assumes the direction is forward (>) with the cursor located at or before the start of the sample sentence. In the sentence `That's my hat!`, a token replacement for `hat` with `umbrella` replaces the last word `hat` in the sentence with `umbrella` whereas a literal replacement would substitute `umbrella` for the `hat` imbedded in `That's` (resulting in `Tumbrella's my hat!`).

Direction applies in the Replace command and is shown by the first character in the command's prompt. If the direction is forward (>), the replacement occurs between the cursor position and the end of the file; if backward (<), between the cursor and the beginning of the file.

Repeat factors are available for the Replace command but must be typed before the command is initiated (before R is pressed). A repeat factor causes that number of substitutions to be made. If not specified, the repeat factor defaults to 1. A slash character (/) may also be used to change all occurrences of the specified string in the current direction. The repeat factor (or slash character) is displayed in brackets [ ] in the command's prompt. The repeat factor works differently when the Verify option is used.

The Verify option lets you choose whether or not to make a particular replacement. The combination of a repeat factor with Verify allows you to replace only certain occurences of a string in the file. For example, after pressing 2 R V and typing in the target and substitute strings, the Editor moves the cursor to the first occurrence of the target string and prompts:

```
>Rpl[2]: <sh-exc> aborts,R replaces,' ' doesn't
```

To confirm the replacement, press R. To skip to the next replacement (if any), press the space bar. While using Verify, pressing SHIFT-Select (SHIFT-EXECUTE) aborts the operation and retains all replacements made up to that time.

The "same" option is available with Replace and refers to either the most recent target string (used in a Find or Replace) or the most recent substitute string (used only in Replace). Which string it signifies (target or substitute) depends on where it is used in the Replace command. To use "Same", simply press S in place of the delimited string. If you type S followed by a delimited string, the most recent target is replaced with the specified string. If you type a delimited string followed by S, the specified target is replaced with the last substitute. Both strings may be specified by typing S S. The current assignments of the "same" patterns can be seen by pressing S E (see the Set command for more details).

Note

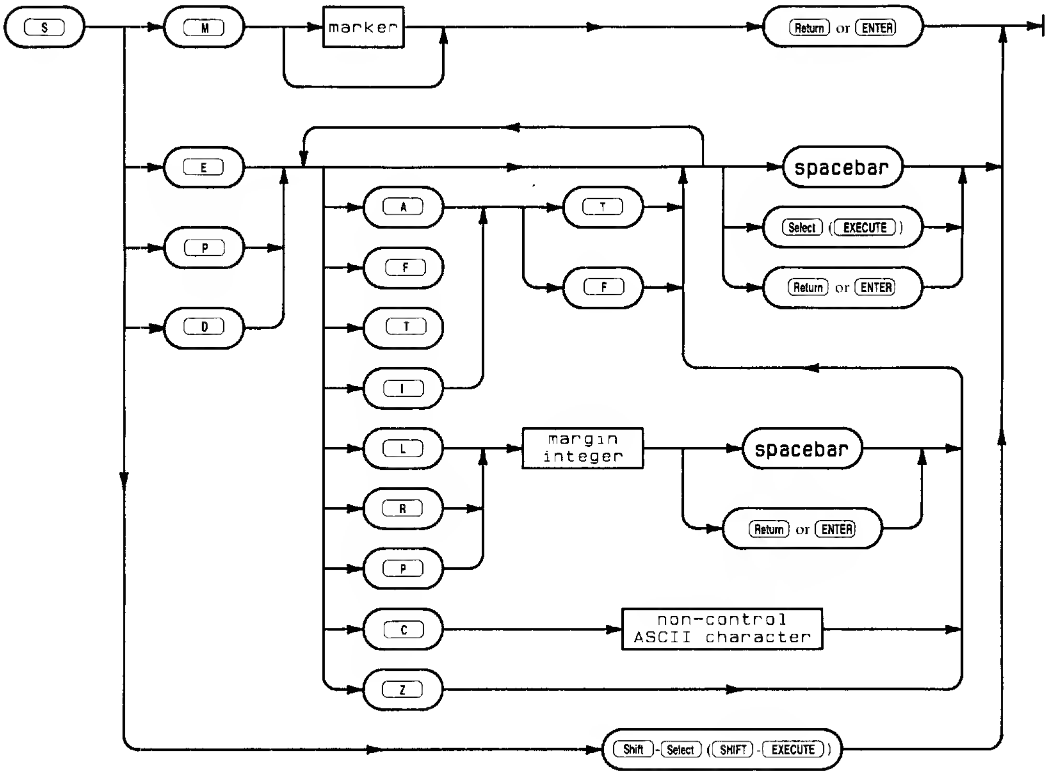
If a Find has been done since the last Replace, the target string used by the "same" option is now the target specified in the Find command.

The "Ignore case" option applies to the Replace command. Type S and E to Set the Environment. Type I and T to set Ignore case to True. Type T and T if Token is not already True. The target string and all **token** strings in the text are treated as upper case. When a match is found, the token string is replaced with the substitute string. The case of the substitute string is not affected by the Ignore case option.

The Replace command can be aborted before all specifications are complete by pressing SHIFT-Select (SHIFT-EXECUTE). (Subsequent use of the "same" option after aborting the Replace command may give you unwelcome results).

SET

Set defines markers and alters the environment in which your text operations occur.



Item	Description/Default	Range Restrictions	Recommended Range
marker	literal	1 to 8 ASCII characters excluding CHR(0) thru CHR(31) and CHR(127)	-
margin integer	integer numeric constant	0 thru 9999; left margin must be less than right margin	0 thru 49 for 50-column displays; 0 thru 79 for 80-column displays
non-control ASCII character	literal	any valid ASCII character excluding CHR(0) thru CHR(31) and CHR(127)	-

Semantics

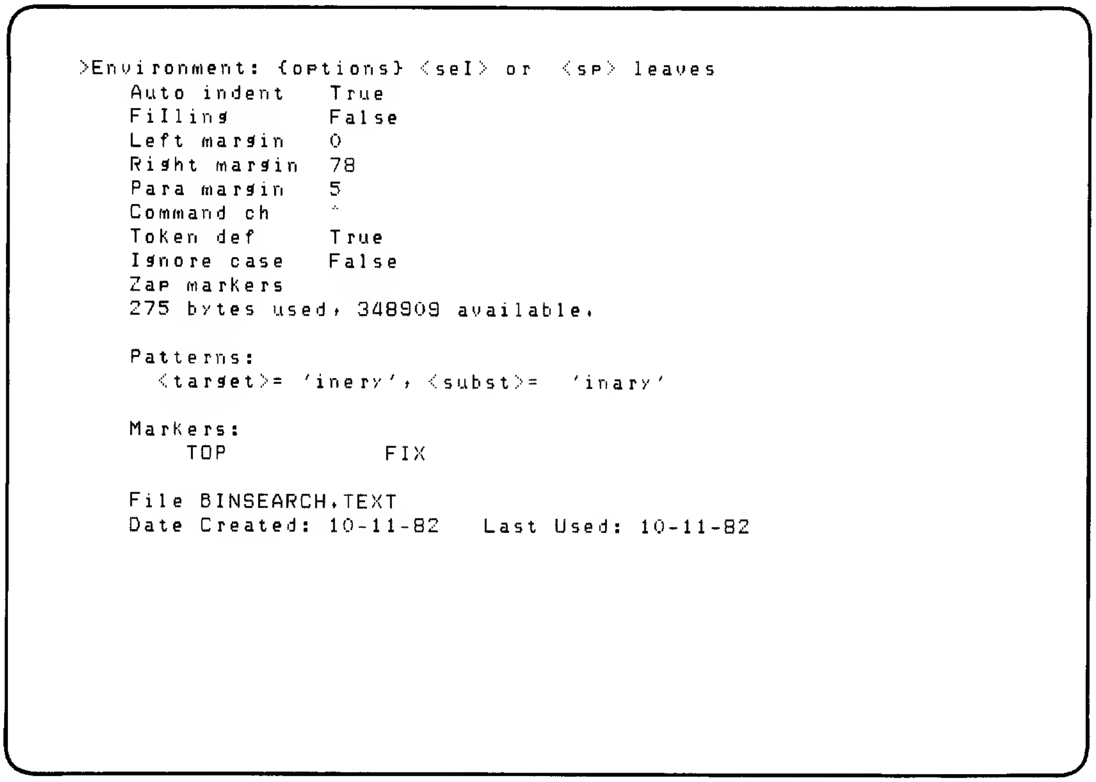
The Set prompt:

```
>Set: Env Mrk Prog Doc <sh-sel>
```

Set lets you define markers and various environment parmeters. Markers are Set by moving the cursor to where you want the marker, pressing S M (for Set Marker) and typing in a marker name followed by Return or ENTER. A marker name is any sequence of up to eight non-control ASCII characters. The Editor accepts more than eight characters but truncates anything longer. All non- printing characters (those with an ASCII value of either 127 or in the range of 0 to 31) are deleted by the system. The Editor converts these names to uppercase so they can be typed in whatever form is convenient.

No more than ten markers can be set in a file. If you attempt to set more than ten, the Editor displays the markers in a numbered list and prompts you for the number of the marker you wish to replace. All markers can be removed by giving the Zap marker command. Markers are used with the Jump and Copy commands and their names are shown in the environment display. The locations of the markers are not shown so the use of meaningful marker names is advised.

Pressing S E (for Set Environment) displays the current environment and allows you to change the environment's parameters. When entering the Editor with a new file, the default environment is the Program environment which looks like:



```
>Environment: {options} <sel> or  <sp> leaves
    Auto indent    True
    Filling        False
    Left margin    0
    Right margin   78
    Para margin    5
    Command ch     ^
    Token def      True
    Ignore case    False
    Zap markers
    275 bytes used, 348909 available.

    Patterns:
      <target>= 'inery', <subst>=  'inary'

    Markers:
       TOP           FIX

    File BINSEARCH.TEXT
    Date Created: 10-11-82   Last Used: 10-11-82
```

Patterns and Markers are only shown if they have been set. The heading near the bottom displays a file name if the Editor is entered with a specified file. Whenever a file is saved on a mass storage medium, the current environment is saved with it and becomes the default environment when that file is used by the Editor.

The environment also displays how many bytes of memory have been used and how many are still available for use in the Editor. The total number of bytes (used and available) depends on the amount of memory in your machine.

To change a parameter in the environment, press the first letter in the parameter's name. The cursor is automatically positioned at the item to be changed and the new value must be typed. If the parameter needs a number (as in Left, Right and Paragraph margins), then the number must be followed by pressing (Return) or (ENTER) or the space bar. All other parameters accept a single character and return the cursor to the environment's prompt as soon as the character key is pressed.

Automatic indenting is a boolean (with either a true or false value) which affects the Insert and Margin commands. When inserting text with this item set true, pressing (Return) or (ENTER) automatically moves the cursor to the next line at the same starting column as the previous line. This indenting feature is useful when writing Pascal programs so it is set true for the Program (default) environment.

When Auto indent is true the Margin command is disabled. When Auto indent is false, pressing (Return) or (ENTER) places the cursor on the next line at either the Left margin or Paragraph margin (as currently defined in the environment).

Filling is another boolean value which affects the Insert and Margin commands. It usually has a value opposite that of Auto indent. When set true (and auto-indent is false), filling causes automatic "wrap around" of text. If a word is too long to fit on the current line (as defined by the Right margin value), it is carried or wrapped around to the next line and no carriage return ((Return) or (ENTER)) is necessary. Another effect of this parameter being set true is that an Insert completed by pressing (Select) ((EXECUTE)) causes all text following the insertion in that paragraph to be margined or filled according to the current values of the Left, Right and Paragraph margin settings. All blanks in the text are then compressed to a single blank (though two blanks are allowed following the characters: ? . ! :). The Margin command only works when Filling is set true and Auto indent is set false.

With Filling set false, the wrap around and margining functions are disabled. When approaching the end of a line, the system generates a "beep" to inform you that you need to press (Return) or (ENTER) to go to the next line. If you type past the display area of the screen, an exclamation point (!) is shown in the last column. The text, though not visible, is maintained in the computer's memory.

The Left margin may be set to any integer between 0 and 9999. Numbers longer than 4 digits are truncated by the system. The Left margin must be less than the Right margin setting or an error message is generated when you attempt to exit the environment.

The Right margin setting has the same numerical limitations as the Left margin. Unless you have a particular reason for doing so (like making full use of a 132 column printer), it is not a good idea to set this margin beyond the right column display limits of your screen because the text will not be visible.

The Paragraph margin can be set to any positive integer up to 4 digits. This setting determines the indention that the first line in each "paragraph" will get. This occurs when Filling is set false (while inserting text) or when Margin is used. Note that a paragraph as defined by the Editor is any text surrounded by blank lines or by lines beginning with the Command character (discussed below). The beginning and end of a file will also delimit paragraphs.

The Command character can be any non-control ASCII character. If this character is the first non-blank character in a line, the Margin command treats the line as if it were blank. The line is not margined and it is considered to be the beginning or the end of a paragraph. The default Command character is the (^) character.

Token is a boolean used by the Find and Replace commands. When Token is set true, the default value for Find or Replace becomes token and the command's prompt displays the literal option. (Token and literal refer to the type of target string searches that take place in these commands). Conversely, if Token is false, the default value for Find and Replace is literal and the command's prompt displays token as an option.

The **Ignore case** command affects searches in the Find and Replace commands. If "Ignore case" is left as False, then "string" and "STRING" and "String" are not treated as equal. If "Ignore case" is set to True, they are treated as equal. This only works when the Environment's **Token** mode is True or if you type a "T" before typing the target string.

The **Zap markers** command removes all markers from the file.

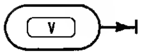
The environment display is left and the Editor's main prompt returned by pressing Return or ENTER, Select (EXECUTE), or the space bar. The current environment settings are automatically saved with your file when the text is written to a disc or other mass storage medium.

Although there is only a single environment associated with a text file, the environment may be set to one of two predefined configurations: the **Program environment** (by pressing S P) and the **Document environment** (by pressing S D). These configurations optimize the various environment parameters for writing programs or regular (non-program and non-tabular) text, respectively. When either predefined environment is Set, the current environment is displayed and any of its parameters can be changed. If you want to change just one or two parameters, use S E to get into the existing environment.

Changes made to the environment cannot be aborted but the parameters may be changed as many times as desired.

VERIFY

Verify refreshes the screen display from memory.

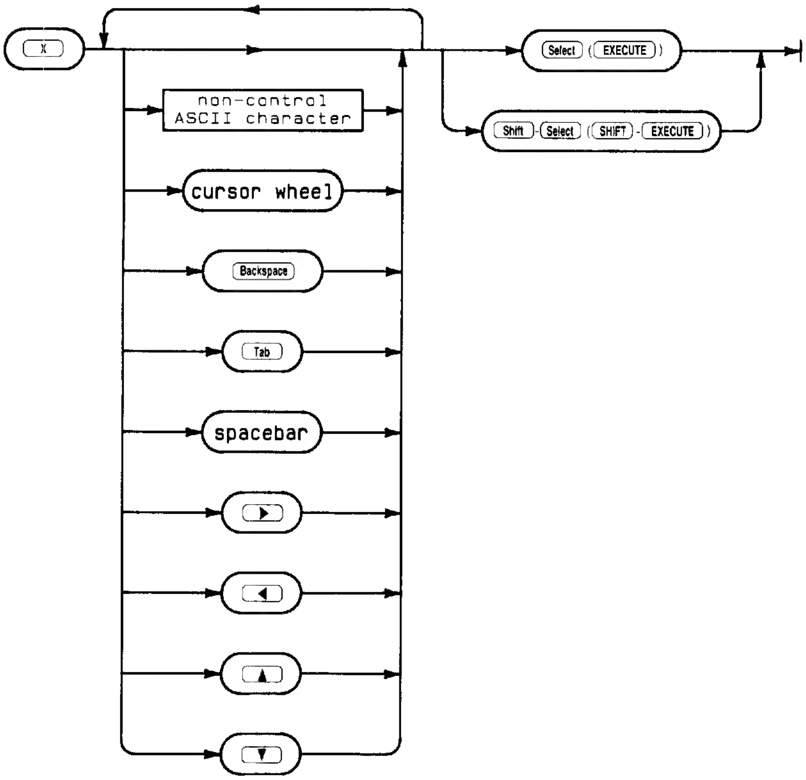


Semantics

The Verify command has no options; it is executed immediately by pressing V. Verify causes the Editor to refresh or update the current screen display from memory, move the current line (the one where the cursor is) to the middle of the screen, and display the Editor prompt. If the cursor is located in the first 23 lines of text when Verify is used, the line containing the cursor is not moved.

eXchange

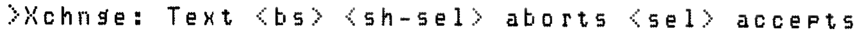
eXchange replaces text character for character at the cursor position.



Item	Description/Default	Range Restrictions
non-control ASCII character	literal	any valid ASCII character excluding: CHR (0) thru CHR (31) and CHR (127).

Semantics

The eXchange prompt:



```
>Xchnge: Text <bs> <sh-sel> aborts <sel> accepts
```

The eXchange command lets you exchange the text in a line on a character-for-character basis, beginning at the cursor position. Typing a character overwrites the character at the current cursor position and moves the cursor to the next character to the right. Using the horizontal cursor-positioning keys is also allowed: backspacing (such as with Backspace, ◀, or cursor wheel) moves the cursor backwards and restores the original character; forwardspacing moves the cursor over the existing characters without changing them.

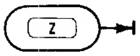
The exchange command operates only on the current line (i.e., the line where the cursor is located when the command is entered). Attempting to move the cursor vertically will generate question marks that overwrite the existing characters. Backspacing past the point at which eXchange was entered is not allowed.

Almost any ASCII character can be used in eXchange; however, use of control characters is not advised. Carriage returns cannot be entered, since you are not allowed to cross line boundaries while in the eXchange mode. Direction and repeat factors do not apply to this command.

eXchange is initiated by pressing X and is exited by pressing Select (EXECUTE) or SHIFT-Select (SHIFT-EXECUTE). Select confirms the exchanges, returns the Editor prompt, and displays the cursor at its position when Select was pressed. SHIFT-Select returns the copy of the text file in the computer's memory to its state before eXchange was entered, displays the Editor prompt and shows the cursor at its position when eXchange was entered.

Zap

Zap deletes text and fills the copy buffer with the deleted text.



Semantics

The Zap command has no options; it is executed immediately by pressing Z. Zap deletes all text between the "anchor" and the current cursor position and stores it in the copy buffer. The anchor is located at the position in the text where the most recent Adjust, Find, Insert or Replace command was executed. (You can confirm the position of the anchor with the Equals command, which moves the cursor to the anchor).

If more than 80 characters are going to be Zapped, the Editor displays a prompt asking if you wish to Zap anyway. Also, if the Copy buffer is not large enough to store the deletion, a prompt asks if you wish to go ahead and Zap the text. (Use the Set environment command to see how much memory is available; the copy buffer shares this memory with that used to hold the text file in memory).

Recovery of the deleted text is achieved with the Copy (from buffer) command. Zap can also be used to move large chunks of text from one location to another within a file.

Note that the effects of Zap can be surprising since the anchor position is set by four different and commonly used commands (listed above). Therefore, it is a good practice to check the location of the anchor (using the Equals command) before executing a Zap.

The Filer

Chapter 4

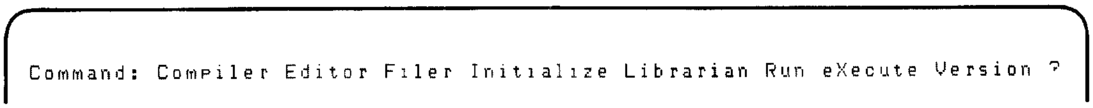
Introduction

This chapter documents the use of the Pascal Filer, a subsystem of the Pascal system. The Filer lets you manipulate files in various ways including moving, listing, duplicating, creating and deleting files. The Filer can handle files on devices with a variety of directory structures and physical characteristics. Before you read this chapter, you should read Chapter 2, Introduction to the File System, which defines basic concepts such as files, volumes and directory organizations.

There are four main sections in this chapter. The first two demonstrate how to enter and use the Filer by leading you through a sample Filer session which uses the more common Filer commands. The next section ("A Closer Look") presents detailed information about the Filer and its operation. The section "Filer Commands" contains an overview or summary of all the Filer commands (useful for quick reference once you are familiar with the Filer) and a semantic and syntactic description of each Filer command in alphabetical order. Any questions you have about commands covered in the sample session should be answered in the commands section.

Entering the Filer

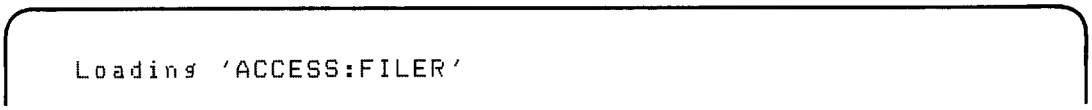
If your system is not already "up and running", refer to Chapter 1 for information on loading the Pascal System. The following prompt must appear on the top line of your screen before you can enter the Filer:



```
Command: Compiler Editor Filer Initialize Librarian Run eXecute Version ?
```

The prompt tells you that you are at the system's Main Command Level — the level from which all the Pascal subsystems (Compiler, Editor, Filer, etc.) are entered. Entry is accomplished by typing the uppercase character of the subsystem you wish to enter (for instance, R for Run and X for eXecute).

Insert the disc labeled ACCESS: and press the F key. You can use either uppercase or lowercase letters when typing commands at the Main Command Level. However, letter case is important when typing file names. The screen displays:

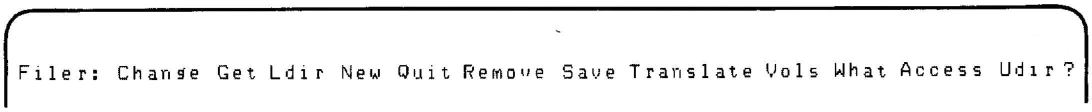


```
  Loading 'ACCESS:FILER'
```

You can use the Permanent command (from the Main Command Level) to keep the FILER code file in memory if you wish. This will allow faster access to the Filer but uses more memory. Chapter 1 explains how to "permanently load" the Filer.

The Filer Prompt

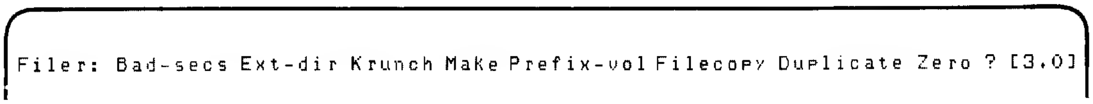
The screen clears and displays the Filer prompt on the top line:



```
Filer: Change Get Ldir New Quit Remove Save Translate Vols What Access Udir ?
```

You are now in the Pascal Filer subsystem. The Filer prompt shows the most common commands used in the Filer and "prompts" you to give the subsystem a command.

The prompt shows only a partial list of the available commands; to see the others, type ?. The prompt line shows the Filer's alternate prompt:



```
Filer: Bad-secs Ext-dir Krunch Make Prefix-vol Filecopy Duplicate Zero ? [3.0]
```

The alternate prompt displays the revision number of the Filer in brackets. Type ? again and the main Filer prompt reappears.

All Filer commands are initiated by typing a single key corresponding to the first character of the command shown in the Filer prompt. Uppercase and lowercase command characters are treated as equivalent by the Filer, so the keys may be typed in whatever form is convenient.

Filer operations can be aborted by typing SHIFT-Select (SHIFT-EXECUTE) when a single character is expected and SHIFT-Select Return or ENTER in place of a file specification.

Filer Operations

All of the commands in the Filer operate in one of two ways: the Filer either performs the operation immediately (when you press the letter key for that command) or it requests the information it needs to perform the operation and then does it. The request is generally for a volume specification or a file specification since all of the Filer's commands (except Quit) operate on volumes, directories and files.

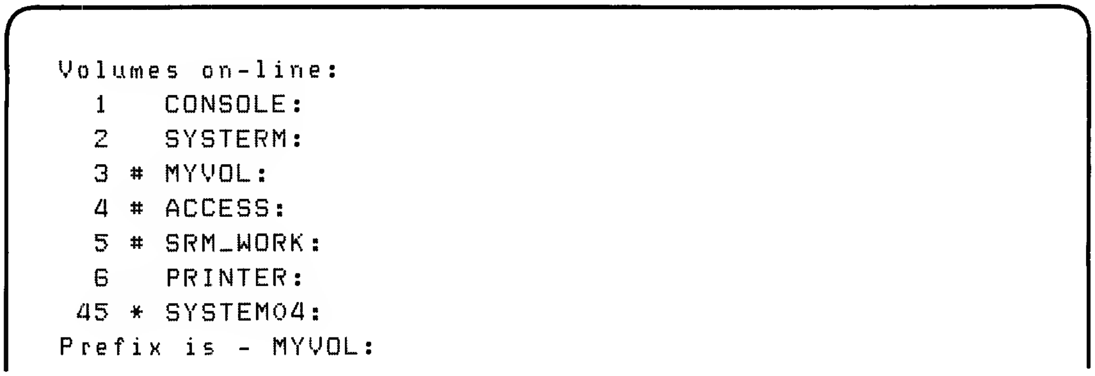
A volume specification identifies a particular volume. This can be done by supplying any of the following: the name of the volume; its associated unit number; a colon (:) to specify the default volume; or an asterisk (*) to specify the system volume. A file specification consists of both a volume specification and a file name; it completely identifies a particular file. All file specifications include a volume specification even if by default. If the volume specification is omitted and only the file name is given, the Filer looks for a file of that name on the default volume.

A Sample Filer Session

Work through the following examples on your machine as you read through this section. Interacting with the computer will teach you more about the Filer than just reading the material.

Finding Out What Devices are Accessible

Now that you have the Filer prompt on the screen, press V. This initiates the Volumes command and the screen now displays the volumes or I/O units associated with the Pascal System. Here is a typical display (yours may vary slightly):



```
Volumes on-line:
  1    CONSOLE:
  2    SYSTERM:
  3 #  MYVOL:
  4 #  ACCESS:
  5 #  SRM_WORK:
  6    PRINTER:
 45 *  SYSTEM04:
Prefix is - MYVOL:
```

For each volume, the display shows the logical unit number and the associated volume which are currently on-line. Volumes #5 and #45 are SRM volumes which you may or may not have.

The "#" beside units 3, 4 and 5 indicates that these are blocked devices. These are used for mass storage. The "*" beside unit 45 indicates that this is the system volume; also a blocked device. The system volume is used by the system during certain operations and should be left on-line at all time if possible. "Prefix is" indicates which is the default volume. The default volume is assumed when no volume identifier is given. The default volume can be changed using the Filer's Prefix command and the Main Command Level's What command.

The Default and System Volumes

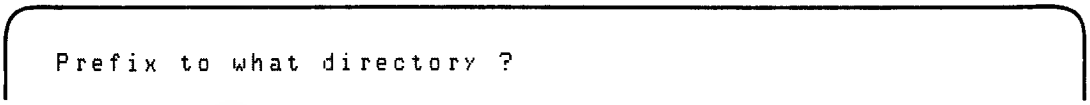
At power-up, the system generally designates the highest performance mass storage device as the "system volume" because it needs a volume to work with occasionally. The system volume is denoted by an asterisk (*) in the Volumes command display and remains fixed unless the New system volume command or the What command is used at the Main Command Level.

The Prefix command, because it defines the default volume, lets you specify a particular volume where the Pascal System will look for files when you haven't given a volume name or logical unit number. This is handy in the Filer as well as in other subsystems such as the Compiler or Editor.

The default volume can be indicated with the colon (:) character. For example, to list the directory of the default volume, press L (for the List directory command) and answer the prompt by typing ":". The Filer then displays the directory of the current default volume.

Changing the Default Volume

You can use the Volumes command (from the Main Command Level) to see what volume is the current default volume. It is listed under the heading `Prefix` in the Volumes display. You can also see what the current default volume is by pressing P for the Prefix command. The screen displays:



```
Prefix to what directory ?
```

Respond by pressing Return or ENTER. The screen now displays the current default volume. Now press P again and in response to the prompt, type `MOJO:`. The screen now displays:

```
Prefix is MOJO:
```

Now, whenever you want to specify a file or group of files on the MOJO: volume, you can just type the file name(s) and the Filer will assume that the file or files specified are on the volume MOJO:.

It is possible to set the default prefix to a flexible disc drive, regardless of the volume inside. This is done by typing:

`#3:` Return or ENTER

while the drive door is open.

The prefix command is used to set up a working directory on a Shared Resource Management system as well. If you had an SRM file named:

```
#5:USERS/JOE/PROJECT1/PROGRAMS/FILE
```

Initiate the Prefix command as usual and specify:

```
#5:/USERS/JOE/PROJECT1/PROGRAMS
```

This sets the SRM volume as the default volume and USERS/JOE/PROJECT1/PROGRAMS as the working directory on the SRM. If the previous working directory had been PROJECT1, then only PROGRAMS need be typed. Now you can specify the file with:

```
FILE
```

If you were to use the Prefix command again to set the default prefix to another volume (not on the same unit), the working directory and volume name for the unit remain PROGRAMS. You need only specify either of the following to get the same file.

```
#5:FILE
```

or

```
PROGRAMS:FILE
```

It is possible to change the working directory on an SRM unit without changing the default volume. Use the Filer's Unit directory command. Press U and then give the directory name that you wish to become the working directory. If the new directory is in the existing working directory, just type the new directory name. If it is not, type the whole directory path as shown above in the Prefix example.

Note

Do not use the Prefix command on unit #45. This is the system volume and should not be altered.

The System Volume

The system volume can also be specified in a shorthand form using the "*" character. Suppose you want to specify the file named LIBRARY on the volume SYSVOL:. Assuming that SYSVOL: is the system volume and is currently in the disc drive associated with unit #3, you can specify a file on that volume by entering any one of the following three methods:

```
SYSVOL:LIBRARY
```

or

```
#3:LIBRARY
```

or

```
*LIBRARY
```

Of course you can make the specification even shorter by typing something like:

```
*=ARY
```

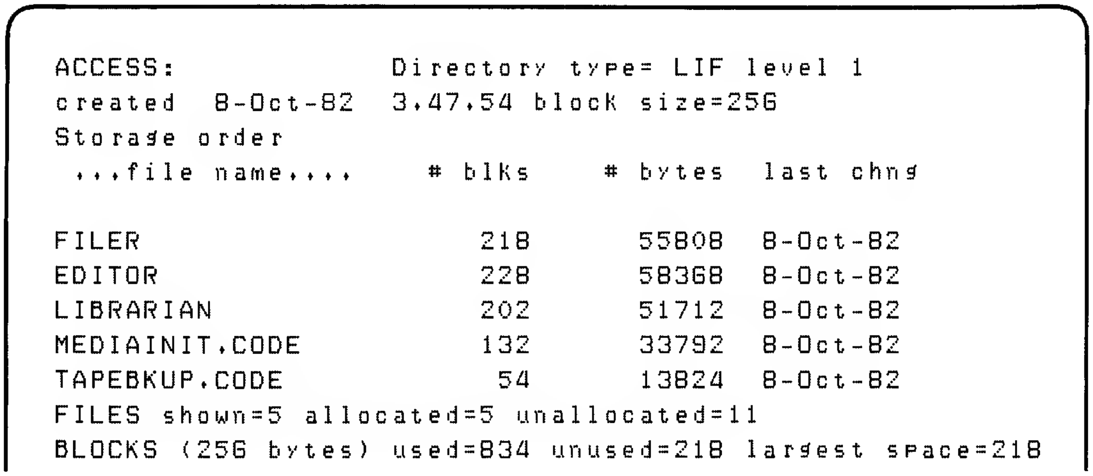
However, if you are doing a critical operation, be sure that there are no other files on the same volume which fit that file specification or use the ? wildcard instead. If a file named GARY existed on SYSVOL:, the operation would also be performed on it. Once again, **use wildcards judiciously**.

Listing a Directory

To find out what files are on the disc called ACCESS:, press L to initiate the List Directory command. The Filer prompts you to specify the volume whose directory is to be listed:

```
List what directory ?
```

Respond by typing `ACCESS:` and pressing Return or ENTER. Notice that the colon (:) is part of the volume specification. The screen now displays the directory (catalog) for `ACCESS:`. It looks similar to this display:



```
ACCESS:            Directory type= LIF level 1
created  8-Oct-82  3.47.54 block size=256
Storage order
 ...file name....     # blks     # bytes   last chng

FILER                   218       55808   8-Oct-82
EDITOR                  228       58368   8-Oct-82
LIBRARIAN               202       51712   8-Oct-82
MEDIAINIT.CODE          132       33792   8-Oct-82
TAPEBKUP.CODE            54       13824   8-Oct-82
FILES shown=5 allocated=5 unallocated=11
BLOCKS (256 bytes) used=834 unused=218 largest space=218
```

The name of the volume is displayed in the upper left-hand corner of the listing. To the right, the directory type is displayed. Pascal discs have Level 1 directories. Level 1 directories contain the date the directory was created and the size of the volume. Level 0 directories do not. Your directory listing should display the date the directory was created and the date it was changed as system volume, the size of the storage blocks, and whether the listing is in storage order or alphabetical order. The size of blocks on LIF volumes is 256 bytes (1 sector). The size of blocks on WS1.0 volumes is 512 bytes. The Shared Resource Management system does its accounting in 1-byte units. To have directories listed in alphabetical order, include [*] after the directory name. For example:

```
LIFDIR:[*]
```

The column entries for each file include: file name, number of blocks used for storage, the file size in bytes, and the date the file was created or changed.

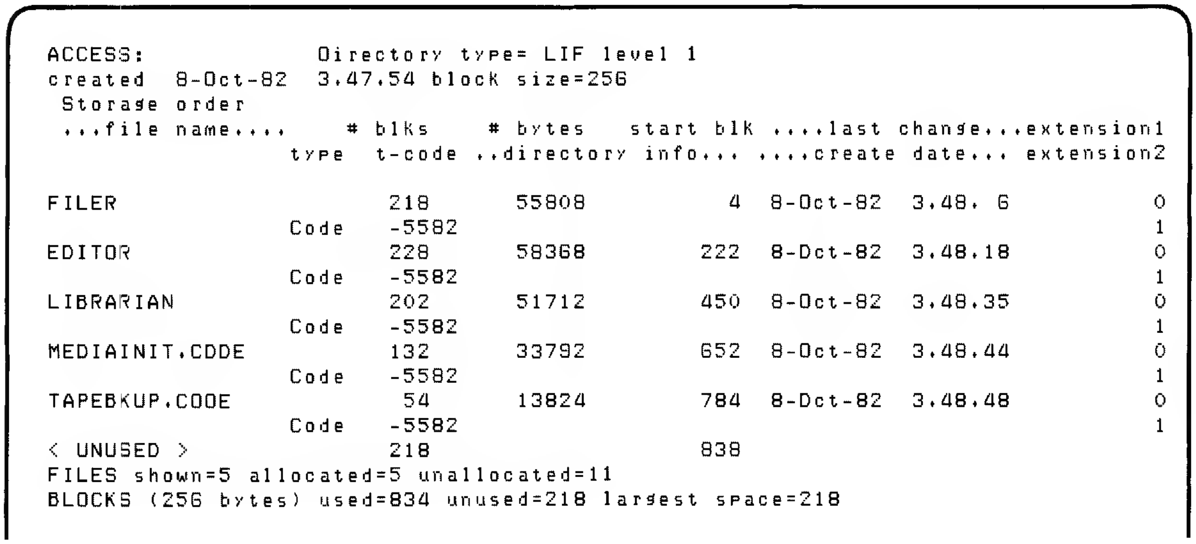
The last two lines display additional directory information including how many more entries can fit in the directory. The size of a directory is specified when the disc is initialized.

Getting a More Detailed Listing

To get a more detailed listing of the directory on a disc, press (E) (for the Extended Directory command) and you will be prompted for a volume name as before. Respond by typing:

`ACCESS:` (Return) or (ENTER)

Your screen now displays:



```
ACCESS:          Directory type= LIF level 1
created  8-Oct-82  3.47.54 block size=256
 Storage order
 ...file name....     # blks    # bytes   start blk ....last change...extension1
                 type  t-code ..directory info... ....create date... extension2

FILER                   218       55808          4  8-Oct-82  3.48. 6          0
                 Code   -5582                                                  1
EDITOR                  228       58368        222  8-Dct-82  3.48.18          0
                 Code   -5582                                                  1
LIBRARIAN               202       51712        450  8-Oct-82  3.48.35          0
                 Code   -5582                                                  1
MEDIAINIT.CDDE          132       33792        652  8-Oct-82  3.48.44          0
                 Code   -5582                                                  1
TAPEBKUP.CODE            54       13824        784  8-Dct-82  3.48.48          0
                 Code   -5582                                                  1
< UNUSED >              218                    838
FILES shown=5 allocated=5 unallocated=11
BLOCKS (256 bytes) used=834 unused=218 largest space=218
```

The Extended directory listing contains all the same information as the List directory listing with additional information. It also contains the number of the block where the file starts, the type as recognized by the file system, the type-code used by the directory system, SRM access information and two extension fields.

The "directory info" column shows the public access rights and the current file status for SRM files. (Because the listing above is not from an SRM volume, the column is empty.)

If one of the letters from the table below is missing, then the public access right associated with that letter has been removed.

Letter	Access Right
M	Manager
R	Read
W	Write
S	Search
P	Purgelink
C	Createlink

Public access rights on a file are established at one of two times. If a file is created by a program, the public access rights can be established when the file is opened. To do this, use the optional third parameter on the command used to open the file. The commands used to open files are Reset, Rewrite, Open and Append. The optional third parameter is explained in more detail in the File System chapter and in the HP Pascal Language Reference.

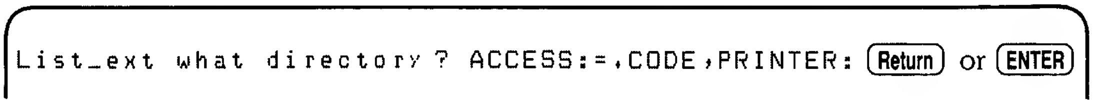
If a file already exists, the Filer's Access command can be used to establish or, if the Manager right has not been removed, change the public access rights.

The possible "current file status" are listed below and explained in the "File System" chapter.

CLOSED
SHARED
EXCLUSIVE
CORRUPT

The two extension fields are for LIF directories only. The other directories display a "–1". For most LIF file types, extension1 contains a "0". For system files, it contains the start execution address. For data files, it contains the logical end-of-file. Extension2 contains the volume number in cases of multi-volume files. The Pascal system cannot create or read multi-volume files; the LIF DAM merely recognizes them. For single volume files, it contains a "1".

The above examples are the most common uses of the directory listing commands, but there are two other useful ways of using the command. One is to use a "wildcard" to specify a subset of files that you want listed. The other way is to send the listing to the printer or to a file instead of letting the listing default to the screen. Both methods are combined in the example below and are covered in detail in the "Filer Commands" section. Press E again (to initiate the Extended Directory command) and answer the prompt for a volume specification as shown in the display:



```
List_ext what directory ? ACCESS:=.CODE,PRINTER: Return or ENTER
```

The ACCESS: volume should be on-line. Your specification tells the Filer you want a listing of all the files on the ACCESS: disc whose name ends in ".CODE". The "=" acts as a substitute for all combinations of characters in a file name. The "," separates the source specification from the destination specification. The listing should only display the files whose names end with ".CODE". The EDITOR, FILER and LIBRARIAN are not listed because their names don't end in ".CODE".

A Few Words About Wildcards

Wildcards are powerful tools for executing Filer commands on related files. There are three wildcard characters.

```
?          =          $
```

A wildcard is a substitute for an arbitrary portion of a file name. For example, if you wanted to list all the .CODE files on the EXAMP: volume, you could specify:

```
EXAMP:=.CODE
```

The "=" stands for any combination of characters. If the file name ended with ".CODE", that file would appear in the listing. If you wanted to remove some of the .TEXT files on the EXAMP: volume, you could specify:

```
EXAMP:?.TEXT
```

The "?" also stands for any combination of characters. However, the Filer will ask you, one at a time, if you want to remove each file that fits the specification. The "?" wildcard lets you verify operations before actually performing them. Unless you are absolutely certain about the effects of a command using the equals sign wildcard (=), its best to use the question mark — by far the safer of the two.

The "$" character is a valid wildcard for destination file specifications. It indicates that the file is to retain its original name. If "$" is used with other characters, it is used as part of the name.

Wildcards act as replacement strings in file names. Part of a file name can be given before or after the wildcard or both before and after. For example, two files named WILD.TEXT and WILD.CODE on the default volume could be specified by:

`WILD?` or `WILD=` or `=LD.=` or `?ILD=`

Partial file names must be given in the order in which they appear in the file name.

Translating Text Files

The Pascal system supports several different types of "text" files. These files are usually created by the Editor and can be programs, documents, or data. When the file is stored on a disc, the internal representation (format) of the file is determined by the suffix appended to the file specifier. The different formats have different information in the file header and can have different end-of-line schemes. The Translate command can be used to convert from file type to another. The various file formats recognized by the Pascal system are TEXT, ASCII, and DATA. No suffix indicates a DATA file, a .TEXT suffix indicates a TEXT file and a .ASC suffix indicates an ASCII file.

To use the Translate command, press T and see the prompt:

```
Translate what file ?
```

Respond with the name of your input file

```
MYVOL:MYFILE.TEXT
```

The Filer will then prompt:

```
Translate to what ?
```

Respond with the name of your output file

```
MYVOL:NEWFILE.ASC
```

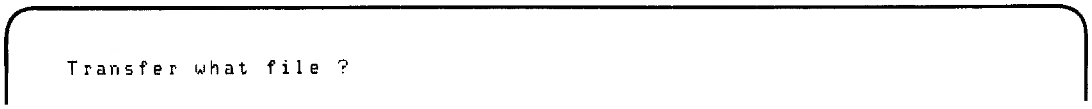
The Filer will create an output file of a type corresponding to the suffix on the output file name (ASC in the example) and will read the text data from the input file, reformat the data to match the output file type, and write the data to the output file. This process may seem slow, but remember that the text is being reformatted.

Sending File Listings to the Printer and Screen

The Translate command is used to send files to the printer or to the screen. Logically, the printer and screen are just files of a different format.

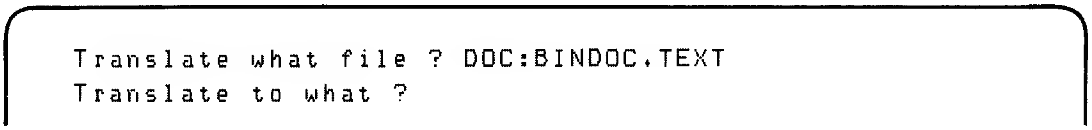
Before using the Translate command, remove the volume ACCESS: and replace it with the volume DOC: (supplied with this manual set). Now use the Extended Directory command to display the contents of the DOC: volume. Press E, type in DOC: and press Return or ENTER. Your screen should display all the files on the documentation disc.

Press the space bar: this clears the screen of everything except the Filer prompt. Now press T to initiate the Translate command. The screen prompts you with:



```
Transfer what file ?
```

Respond with `DOC:BINDOC.TEXT` and press Return or ENTER. The screen now prompts:



```
Translate what file ? DOC:BINDOC.TEXT
Translate to what ?
```

Your first response includes both a volume specification and a file name and completely identifies the file you want to transfer. Now type `PRINTER:` and press Return or ENTER. The text file is translated to the printer as shown:

```
BEGIN {Binery_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN          {This is the actual binery search}
 mid:= ROUND((top + btm)/2);
   IF key = alpha[mid] THEN done:= TRUE
   ELSE IF key < alpha[mid] THEN top:=mid
   ELSE btm:=mid;
   IF top=btm THEN BEGIN
                     done:=TRUE;  mid:= -1;
                   END;
 END;
 IF mid > 0 THEN
  WRITELN('Key -',key,'- is in position ',mid:2)
 ELSE WRITELN('Key - ',key,' - was not found');
END.
```

The Filer shows you what operation it has just performed by displaying:

```
DOC:BINDOC.TEXT                    ==> PRINTER:
```

Since the operation is complete, the Filer again displays its prompt. Note that only files of type TEXT, ASCII or DATA should be sent to the printer. You can also Translate these files to the screen by using CONSOLE: in the destination specification instead of "PRINTER:". The file is displayed one screen at a time. Press the spacebar to move to the next screen; press SHIFT-Select (SHIFT-EXECUTE) to abort the operation.

If you are not sure if the file in question is a text file, use the Extended Directory command and look at the column in the display where the file types are shown.

Copying Entire Volumes: Backup Copies

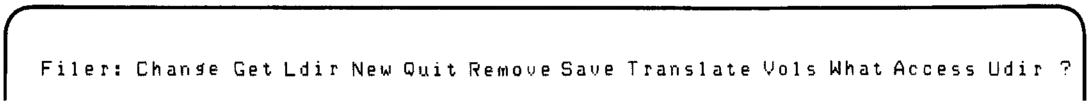
The backup process described here is suitable for volume-to-volume copies if both volumes are the **same** size. For **different** size volumes, see Filecopy in the "Filer Commands" section.

Note

Using Filecopy to copy an entire volume will result in the loss of disc space if the source volume is *smaller* than the destination volume. To copy a volume to a larger one, Filecopy individual files.

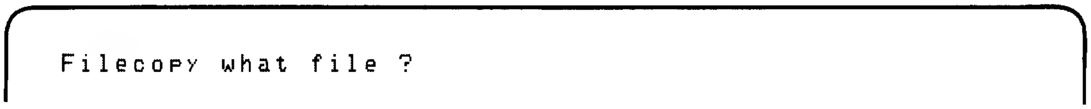
You should still be at the Main Command Level and now have a blank initialized disc. We will use it for a volume-to-volume filecopy. Volume-to-volume filecopies do not require that a directory be present on the destination disc.

Insert ACCESS: in the disc drive. Press F and the Filer gets loaded and displays its prompt:



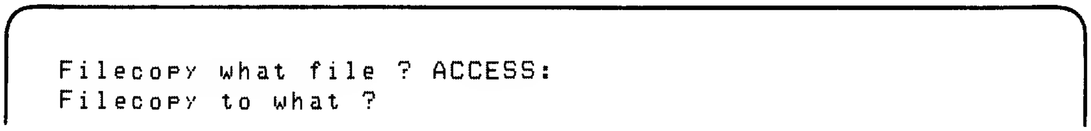
```
Filer: Change Get Ldir New Quit Remove Save Translate Vols What Access Udir ?
```

Press F for the Filecopy command and the screen shows:



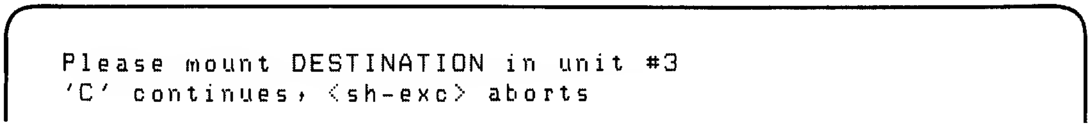
```
Filecopy what file ?
```

Now type `ACCESS:` and press Return or ENTER. The screen displays:



```
Filecopy what file ? ACCESS:
Filecopy to what ?
```

You can specify a volume by specifying the logical unit number associated with the physical disc drive that it is in. Do this by typing #3: and pressing Return or ENTER. The Filer knows that ACCESS: is currently in the drive associated with unit #3 and figures that you want to transfer that volume to a different volume that will be inserted in the same drive. The Filer then reads as much of ACCESS: as it can into read/write memory and the screen displays:



```
Please mount DESTINATION in unit #3
'C' continues, <sh-exc> aborts
```

Now remove ACCESS:, replace it with the blank initialized disc, and press C. Since no directory is on the initialized volume, the Filer simply copies the ACCESS: information that it read into memory onto the new disc. If there had been a directory named TESTER: on the destination volume, the Filer would have prompted:

```
Destroy directory of TESTER: ? (Y/N)
```

This precaution makes sure the information on the disc does not get destroyed if you change your mind or inserted the wrong disc. Answering with N for "No" aborts the Filecopy operation and the Filer prompt returns. Answering with a Y for Yes lets the Filecopy take place and the contents of ACCESS: are written to the new disc. This operation destroys the directory (and, effectively, all information) that was previously on the destination disc.

In case your machine does not contain enough memory to read in the entire volume ACCESS:, the Filer prompts you to swap the source and destination discs as many times as necessary to complete the Filecopy operation. When the operation is complete the Filer prompt reappears.

If you have more than one disc drive you can accomplish the same task by specifying both the source and destination volumes with either a volume name (if it has one) or by the unit number associated with the drive it is in. This second method of doing volume-to-volume transfers is quicker — especially if the amount of memory in your machine is relatively small.

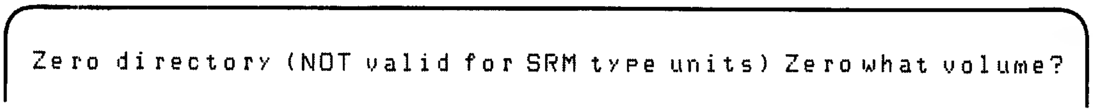
Note

Having two volumes with the same name on-line at one time is not advised. The Filer looks for volumes according to their volume names and may not be able to distinguish one from the other. Thus, the Filer may perform an action on one volume when you wanted the operation to affect the other volume. The Filer warns you whenever it detects that this condition exists. If you get a warning, either remove one of the volumes or use the Filer's Change command to change the name of one of the volumes.

Creating a Directory

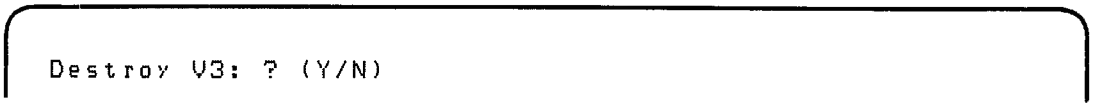
In general, the Filer only works with volumes that already have directories. There are a few exceptions to this rule, such as volume-to-volume transfers where the directory from the source volume is copied onto the destination volume. Other exceptions are mentioned as they arise. The Filer's Zero command creates an empty directory on a new disc that has been initialized using the MEDIAINIT program, previously used discs, or on any other compatible type of mass storage device such as a hard disc or a volume stored in read/write memory. The Zero command, however, is not used to create directories on the Shared Resource Manager. This is done with the Filer's Make command because making an SRM directory really involves making a file of type "directory".

Your screen should now display the Filer prompt. Remove the current volume from the disc drive associated with unit number 3 and replace it with the second disc that you initialized. Now press Z to initiate the Zero command. The screen displays:



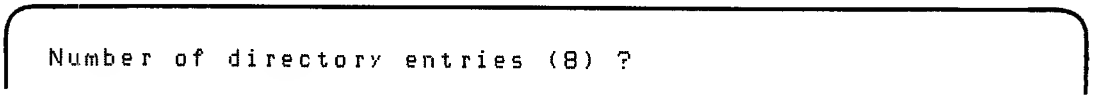
```
Zero directory (NOT valid for SRM type units) Zero what volume?
```

The request is for a volume specification. Answer with `#3:` and press Return or ENTER. The Filer now prompts:



```
Destroy V3: ? (Y/N)
```

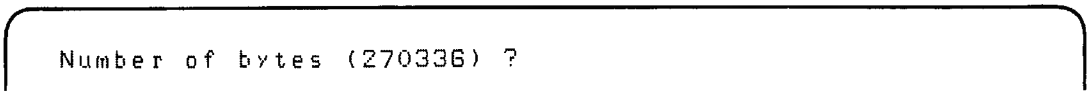
This question is just a safety precaution so that you won't destroy a volume full of information by accident. "V3" is the name given to the directory by MEDIAINIT (if created on unit #3). Press Y for yes. The next prompt is:



```
Number of directory entries (8) ?
```

This is asking for the maximum number of files that will be listed in the directory. The number in the parentheses is the default that will be used if no value is given and is derived from the number in the existing directory. In most cases, 80 directory entries is a good choice.

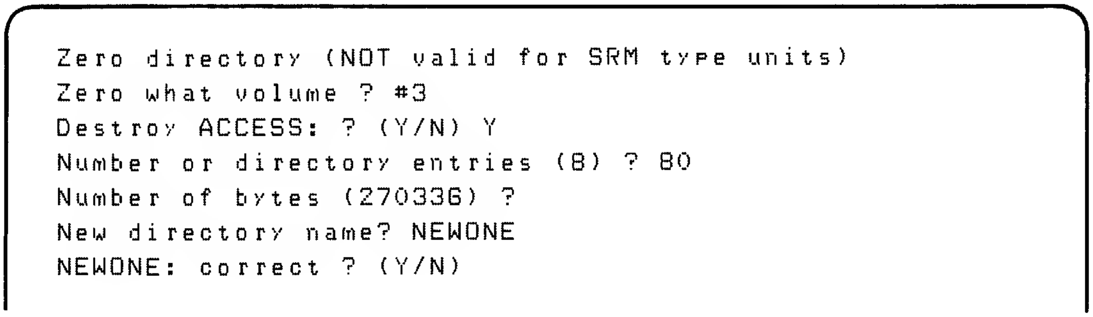
The next prompt is:



```
Number of bytes (270336) ?
```

This is asking for the total size of the disc to be handled by the directory (the logical size of the volume). The number in the parentheses is derived from the number in the existing directory (if any) or from the unit table entry for that given unit. Press Return or ENTER to accept the default size for your disc.

The system now prompts you for a volume name. Volumes and volume name syntax for the different directory types are described in the File System chapter. Briefly, LIF directory names must be six characters or less; upper and lower case characters being distinct. WS1.0 directory names must be 7 characters or less and are always uppercased before being written in the directory. The Filer then confirms that the volume name is the one you wanted. The screen now appears:



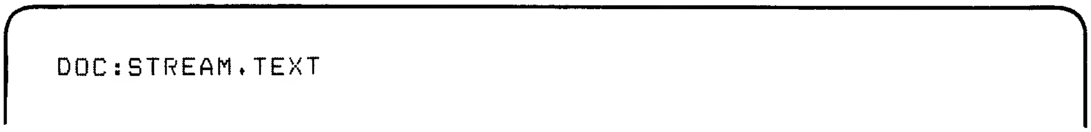
```
Zero directory (NOT valid for SRM type units)
Zero what volume ? #3
Destroy ACCESS: ? (Y/N) Y
Number or directory entries (8) ? 80
Number of bytes (270336) ?
New directory name? NEWONE
NEWONE: correct ? (Y/N)
```

When you press Y to confirm the new volume name the Filer informs you that the volume with that name has been zeroed and the Filer's prompt appears. Your new volume is now ready for use.

Copying Files from Volume to Volume

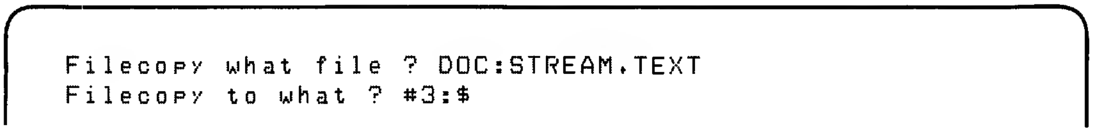
The Filecopy command allows you to copy files from one volume to another or even to a different place on the same volume. The volumes can be separate discs, SRM directories or, in the case of a hard disc, multiple volumes on the same physical device.

Remove the current volume from drive #3 and insert the DOC: volume supplied with this document set. To copy a file from one volume to another, press F for Filecopy and respond to the prompt for a file specification with:



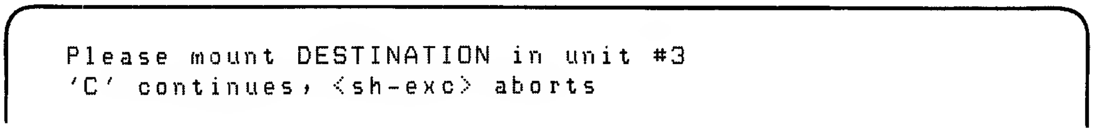
```
DOC:STREAM.TEXT
```

When the Filer prompts you for a destination, type in the specification shown below and press Return or ENTER.



```
Filecopy what file ? DOC:STREAM.TEXT
Filecopy to what ? #3:$
```

What happens here is similar to copying a volume from one disc to another using a single drive. The Filer reads the contents of DOC:STREAM.TEXT into memory and then displays the message:



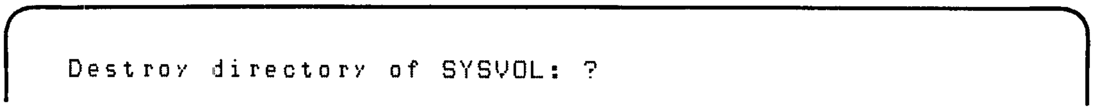
```
Please mount DESTINATION in unit #3
'C' continues, <sh-exc> aborts
```

Take another disc and insert it in drive #3. Now that you have your new disc in drive #3, press C to continue. The Filer writes the contents of the file that it temporarily stored in memory to the disc you just inserted and confirms that the Filecopy has taken place.

If you give a unit number (as above) or a different volume name which is not on-line, you must swap discs to complete the copy.

The wildcard ($) is a feature to avoid repetitious typing and tells the Filer to give the destination file the same name as the original file — `STREAM.TEXT`.

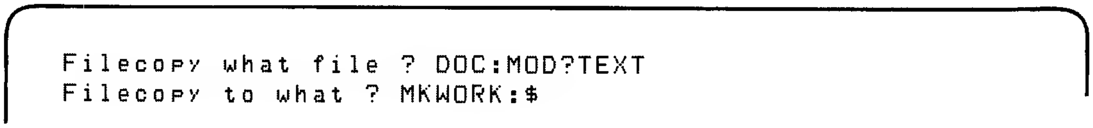
When copying a file to a different volume, **always** include either a file name or the `$` character when you specify the destination. If you specify the name of a mass storage volume without a file name, the Filer prompts:



```
Destroy directory of SYSVOL: ?
```

Although the volume name may be different, if you answer with a Y, the Filer transfers the specified file to the destination volume, **destroying** the directory in the process, and rendering all previous information on that volume useless.

The next example demonstrates how to copy multiple files from one volume to another using the `?` character as a wildcard. Press F once again and respond to both the prompts as shown:

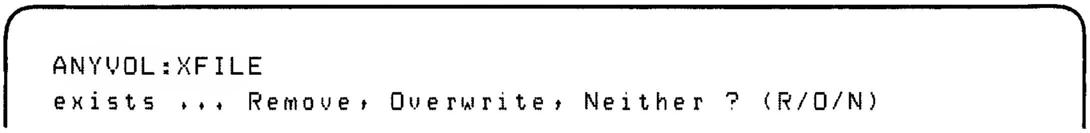


```
Filecopy what file ? DOC:MOD?TEXT
Filecopy to what ? MKWORK:$
```

This tells the Filer to copy all the files on the DOC: volume that begin with the characters "MOD" and end with the characters "TEXT" to the volume MKWORK: and to give them the same name. Before it does this, it will verify with you that you actually want to copy each file that fits the specification. Respond to each prompt with a Y for "Yes" and the three files MODULE_1.TEXT, MODULE_2.TEXT, and MODULE_3.TEXT get copied onto your MKWORK: volume from the DOC: volume. If you have a one-drive system, the Filer will prompt you to swap the discs as in the previous example.

It is worth mentioning that, although specifying a unit number is less typing than specifying a volume name, when you specify a unit number the Filer initially accesses the volume (disc) currently in the drive without regard to whether or not it was the one you intended. After the first access of a volume, the Filer associates a supplied unit number with the name of the volume found in that device. However, if you specify a volume name, the Filer only performs the command on that volume. If the volume you specified is not on-line the Filer will tell you so. Specifying the volume name is a good habit if you are doing a lot of disc "swapping"; this will insure that the Filer does not operate on a disc other than the one you intended to use.

In cases where the destination volume already contains a file with the same name as the file being copied, this prompt is displayed:



```
ANYVOL:XFILE
exists ... Remove, Overwrite, Neither ? (R/O/N)
```

You have the options:

- Remove — remove the original file first, then write the new file in the largest space available.
- Overwrite — replace the contents of the old file with the new information. The Overwrite option cannot be used to change the type of a file on SRM. Attempting to do so will result in the file contents being inconsistent with the file type.
- Neither — cancel the operation.

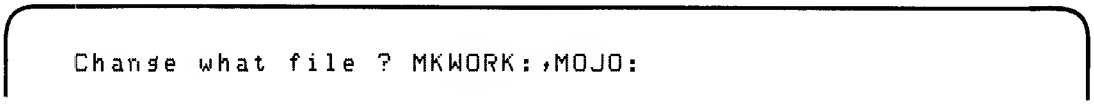
The Overwrite option allows you to put a file in the same starting location as the original. This is important to SRM users when duplicate links exist to a file. All links and passwords to the file are accurate when a file is updated because it is put in the same logical location. If you chose the Remove option, the original file would not actually be removed; only your link to it is removed. The other user's directories are still linked to the original file.

Note

Be careful when using the Overwrite option on an SRM system. If the file specifier suffix (.TEXT, .ASC, or none) is not the same as the original file suffix, the contents of the file may become inaccessable.

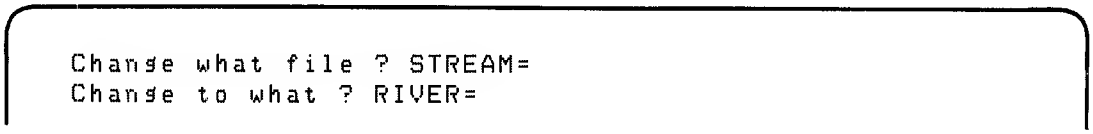
Renaming Files and Volumes

The Filer's Change command allows you to rename files and volumes. (The one exception is that the root directory of the Shared Resource Manager cannot be renamed.) This command requires two specifications: the original name and the new name (the first name may include volume specification and pathname and passwords for SRM, but the second name **cannot**). Assuming that the volume MKWORK: is still on-line, press C for Change and respond to the prompt as shown:



```
Change what file ? MKWORK:,MOJO:
```

The volume name is now "MOJO:". To change the file STREAM.TEXT on the MOJO: volume to RIVER.TEXT you can either type out both names (separated by either a comma or a press of the Return or ENTER key) or use a wildcard as shown below:



```
Change what file ? STREAM=
Change to what ? RIVER=
```

The Filer changes the file name as described. The wildcard was used as a substitute for the .TEXT part of both names. The only restriction on using wildcards with this command is that if you use a wildcard in one of the specifications, you must use it in the other. Because the strings or subsets represented by the wildcard are not always obvious, discretion is advised when using wildcards with the Change command.

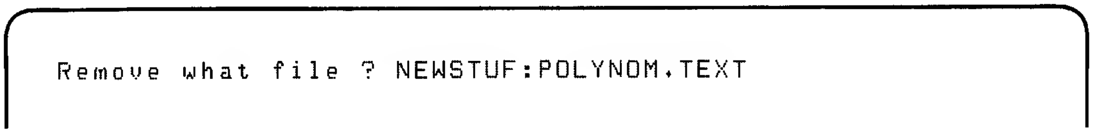
When changing the name of a file of type TEXT or CODE, remember that parts of the Pascal System attempt to append the suffixes ".TEXT" or ".CODE" to the file you specify. You can get around this by specifying a file and adding a period (.) to the file name. This tells the system not to append the suffixes to the file name for which it searches.

Note

Excluding the Get command, the Filer makes no assumptions about sufixes and will treat a trailing period as part of the file name.

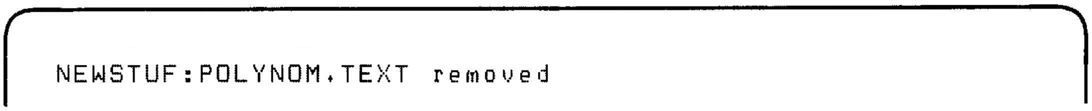
Removing Files

The Remove command is provided to delete files from a directory of a block structured volume. Suppose you have a volume on-line named NEWSTUF: containing the file POLYNOM.TEXT that you wish to delete. Press R to initiate the Remove command and respond to the prompt as shown:



```
Remove what file ? NEWSTUF:POLYNOM.TEXT
```

Then press Return or ENTER. The Filer removes the specified file from the volume and reports:



```
NEWSTUF:POLYNOM.TEXT removed
```

The Filer prompt reappears as the message is displayed. As in many of the Filer's commands, the prompt requests a file specification. Wildcards can be used with the Remove command but should be used carefully. The question mark (?) wildcard provides an easy method for removing a TEXT and CODE file of the same name. It also lets you verify the operation (a good practice when purging files).

Suppose the same volume NEWSTUF: contains two files you wish to remove called IOTEST .TEXT and IOTEST.CODE. To remove these files answer the "Remove what file ?" prompt with:

```
NEWSTUF:IOTEST?
```

and press Return or ENTER. The Filer responds with:

```
Remove IOTEST.TEXT ?  (Y/N)
```

Reply with Y (for Yes) to remove the file. Reply with N (for No) if you change your mind. Either reply results in the next prompt:

```
Remove IOTEST.TEXT ?  (Y/N) Y
Remove IOTEST.CODE ?  (Y/N)
```

Reply as before and the Filer responds with:

```
Remove IOTEST.TEXT ?  (Y/N) Y
Remove IOTEST.CODE ?  (Y/N) Y
Proceed with remove ?  (Y/N)
```

This gives you one more chance to change your mind about the operation. The files are not actually removed from the volume's directory until you press Y. Pressing N has the same effect as if you had never initiated the command (i.e., the directory remains unchanged and your files remain intact).

If you want to remove all of the files on a volume (for discs only; not SRM), the quickest way to do so is to execute the Zero command. This command wipes out the directory of a volume so that the volume may be re-used. See the description of the Zero command earlier in this section or in the "Filer Commands" section.

Leaving the Filer

Exit the Filer by pressing (Q) for Quit from the Filer prompt. You will immediately be returned to the Main Command Level. The Filer can also be exited with the (STOP) key. The (STOP) key waits for any current disc I/O to complete before it actually executes. This key can be used at any time — even while executing a Filer command. However this practice is not recommended since some commands may cause damage to your files if (STOP) is pressed while they are being accessed.

The System Workfile (A Convenient Scratchpad)

The Pascal System features a workfile which can be used in the Filer, Editor, Compiler and Assembler. Each subsystem that uses the workfile documents its use.

Think of the workfile as being analogous to a default volume. In some of the subsystems, you are not prompted for a file specification when entering the subsystem if a workfile of the appropriate type exists. For example, if the text version of a workfile exists when entering the Editor, the Editor never prompts you for a name of the text file to edit but reads in the workfile instead. As a matter of fact, before you can edit any other file, you will need to use the Filer's New command (preceeded by the Save command if you want to retain the file) to release the workfile. In the same manner, invoking the Pascal Compiler when the text version of a workfile exists results in that file automatically being compiled.

If the Filer's Get command is used, the workfile is the TEXT/ASCII/DATA and/or CODE file specified in the command.

The Filer has four commands (Get, New, Save and What) which operate directly on the workfile. These are covered in the next section.

Filer Commands

This section contains a brief overview and summary of the Filer commands and a complete alphabetized description of the syntax and semantics of all the Pascal Filer commands and options.

Filer Command Summary

Volume Related Commands

Bad sectors – Scans a volume and searches for unreliable (bad) storage areas.

Extended Directory – Lists directory information about a specified volume or set of files.

Krunch – Consolidates all unused space on a volume in a single area by packing the existing files together. (Not valid for SRM)

List Directory – Lists directory information about a specified volume or set of files.

Prefix – Specifies a new default volume.

Volumes – Lists the volumes currently on line.

Udir – Sets the default unit directory. (SRM only)

Zero – Creates an empty directory on the specified volume. (Not valid for SRM)

Exit Commands

Quit – Provides an orderly exit from the filer.

STOP – Pressing the (STOP) key unconditionally exits the Filer Subsystem. The current I/O operation is completed before exiting.

File Related Commands

Access – Change the access rights (passwords) on a file or directory. (SRM only)

Change – Change the name of a file, set of files, or volume.

Duplicate link – Duplicates links to a file or set of files. (SRM only)

Filecopy – Copies a file, set of files, or a volume to a specified destination.

Make – Create a directory (SRM) or a file on a volume.

Remove – Remove a directory entry or a set of directory entries.

Translate – Translates text files of types TEXT, ASCII, and DATA to other text file representations or to unblocked volumes.

Workfile Related Commands

Get – Specifies a file as the workfile.

New – Specifies that no file is the current workfile.

Save – Saves the current workfile(s) with the specified name.

What – Lists the name and current state (saved or not saved) of the workfile(s).

Command Syntax and Semantics

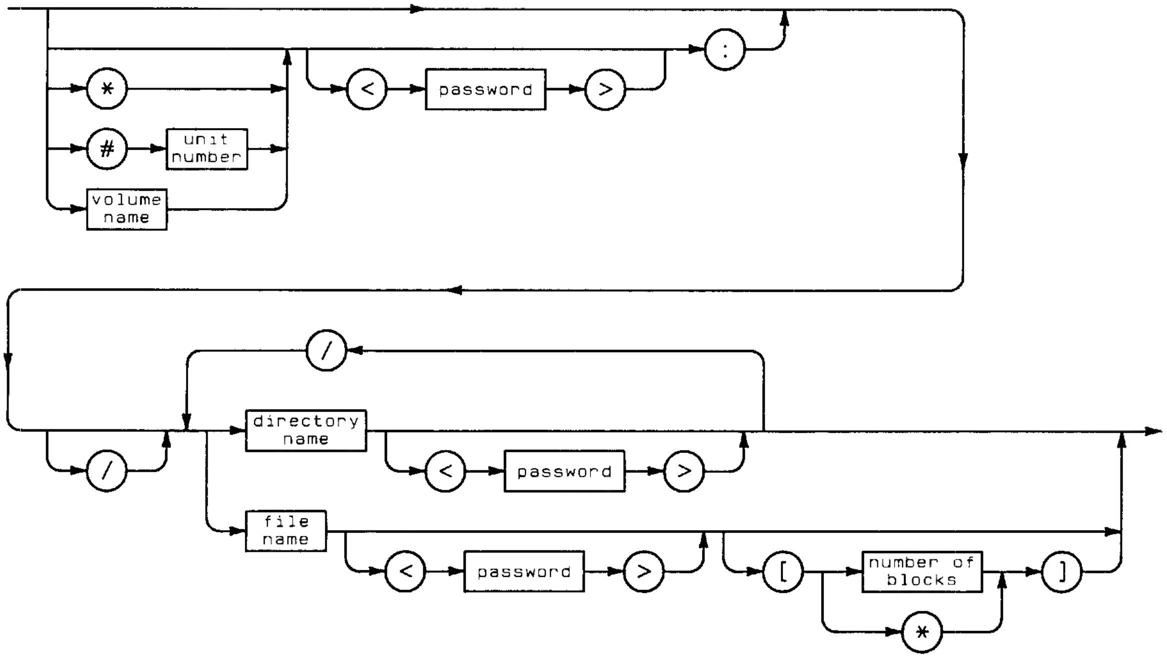
The Filer commands are presented in alphabetical order. Each command's explanation includes: the command's name, a brief functional description, a diagram showing its legal syntax (See Chapter 1), the command's prompt (if any) and text which discusses using the command. Each command's options are also covered and some have examples to show the proper use of these options.

Several of the syntax diagrams on the following pages reference the the "volume only specifier" and the "complete file specifier" below. The "volume only specifier" is the syntax for commands that operate on volumes. The "complete file specifier" is the syntax for commands that operate on files. Volume only specifiers don't need the ":" except when a literal volume name is given. Then the name must end with a ":" to distinguish it from a file name. If no volume specifier is given, the default volume is assumed.

Alphabetical List of Filer Commands

Access
Bad sectors
Change
Duplicate
Extended directory
Filecopy
Get
Krunch
List directory
Make
New
Prefix
Quit
Remove
Save
Translate
Unit directory
Volumes
What
Zero

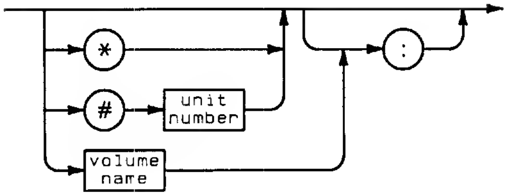
File Specification



Item	Description/Default	Range Restrictions
unit number	integer; corresponding to an entry in the unit table	1 thru 50
volume name	literal	any legal volume name
password	literal	any legal password
directory name	literal	any legal SRM directory name
file name	literal	any legal file name
number of blocks	integer	any legal number of blocks

See Chapter 2 for legal names and values.

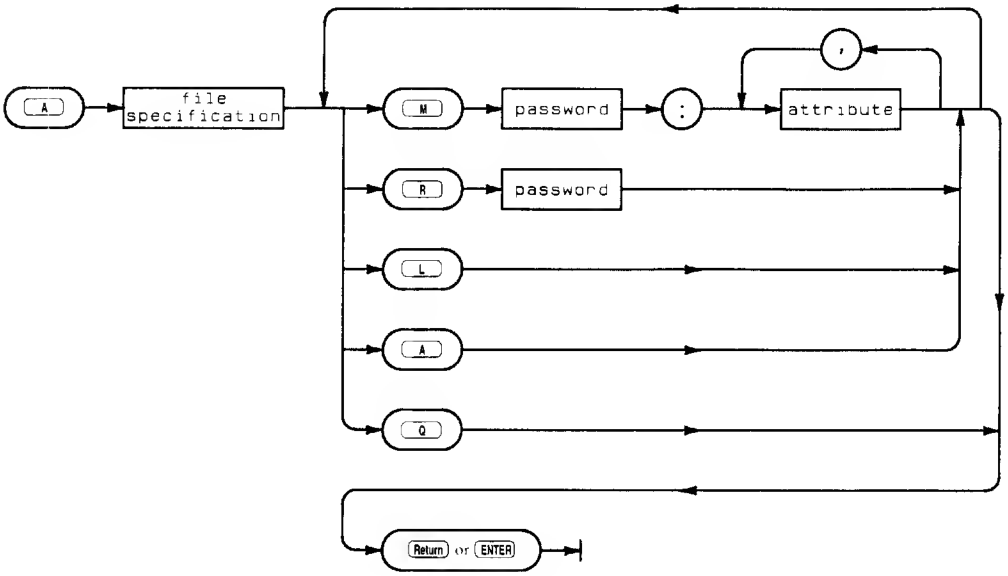
Volume Specification



Item	Description/Default	Range Restrictions
unit number	integer; corresponding to an entry in the unit table	1 thru 50
volume name	literal	any legal volume name

Access

The Access command allows you to change public access rights on your files (SRM only).



Item	Description/Default	Range Restrictions
file specification	literal	a legal SRM file specification
attribute	literal	MANAGER READ WRITE SEARCH PURGELINK CREATELINK ALL
password	literal	any legal password (See Chapter 2 for details)

Semantics

The Access prompt:

```
Access rights for what file ?
```

Type the file specification. If the file already has a Manager password, then you must include the password in the file specification. Access rights cannot be changed on open files or open working directories.

The next prompt:

```
Access: List, Make, Remove, Attributes, Quit ?
```

These are the possibilities. You can list the attribute passwords, make new ones or remove passwords. The Attributes option just lists the possible attributes for your help. Quit returns you to the Filer prompt.

To make new passwords, press M . You see this prompt:

```
Make password:attribute ?
```

Type the password (up to 16 characters), then a colon (:) and the attribute list (attributes separated by commas). Different passwords may be associated with each attribute or one with ALL. If you type a password that already exists, you are asked:

```
PASSWORD already exists...replace it ? (Y/N)
```

Note that passwords should not contain the characters: ">", ":", ",".

To remove passwords, press R . You see the prompt:

```
Remove password ?
```

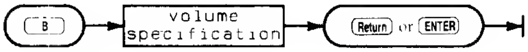
Type only the password and all attributes associated with it are cleared.

The Attributes option list:

```
MANAGER
READ
WRITE
SEARCH
PURGELINK
CREATELINK
ALL
```

Bad sector

The Bad sector command scans a mass storage medium for errors.



Item	Description/Default	Range Restrictions
volume specification	literal	(See the beginning of this section)

Semantics

The Bad sector prompt:

```
Bad sector scan of what directory ?
```

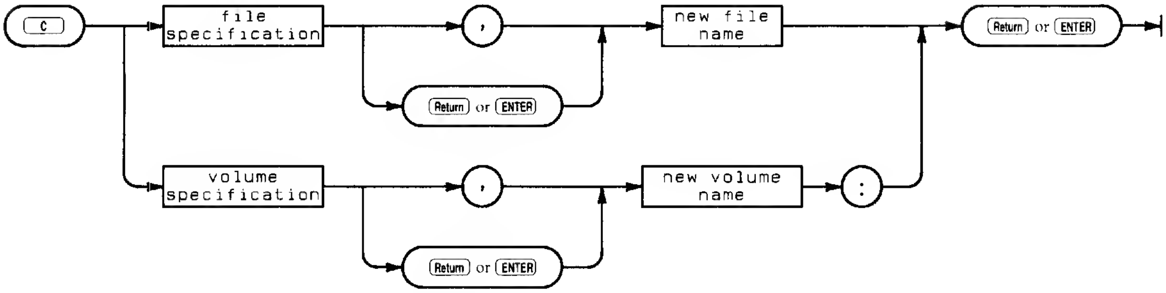
The Bad sector command allows you to check a mass storage medium to find out if each block (sector) is readable. A flexable disc may become unreliable after damage or excessive wear.

Press (B) to initiate the command and answer the prompt with a volume only specifier. The Filer then displays a message indicating that it is scanning the volume from block 0 to the end of the volume. The Filer does a read operation on each sector and does a CRC error check on the results. If the CRC results are normal, that sector is considered to be good; if not, the Filer lists the sector number.

If you find a bad sector in a file, you may wish to use the Filer to change the file type (suffix) to `.BAD`. (You did make a back-up copy didn't you?) The BAD file will not be moved in a Krunch operation. A large number of bad sectors indicates a worn-out medium. The medium should only be used if you are willing to risk losing information on that volume.

Change

The Change command lets you rename files, directories and volumes.



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)
volume specification	literal	(See the beginning of this section)
new file name	literal	any valid file name
new volume name	literal	(See the beginning of this section)

Semantics

The Change prompt:

```
Change what file?
```

The Change command requires two specifications: the original volume or file specification and the new one. The two specifications can be separated by either a comma or a carriage return. If you are changing the name of a volume, any legal volume ID can be used for both specifications.

To change the name of a file, use any legal volume ID in the first specification and only the new file name in the second specification. The Filer is intelligent enough to know that the file whose name you are changing resides on the volume identified in the first specification. After the Filer has finished changing the name and updating the directory it reports the name changes it has made.

Because many of the Pascal subsystems append the string .TEXT or .CODE to a file name given in response to a prompt, it is a good idea to retain these parts of a file name when making a change.

Wildcards (the = and ? characters) may be used in the Change command. If a wildcard is used in the first specification, it must also be used in the second one. The subset string that is replaced by the wildcard in the second specification (the new name) is the same as the string it stands for in the first specification.

Suppose you have a volume named `BUGS:` with the following files:

```
WHATISIT.TEXT
WHOISIT.TEXT
WHYISIT.TEXT
```

Specifying `BUGS:WH=TEXT, FO=FA` in response to the Change prompt results in the following messages being reported by the Filer:

```
BUGS:WHATISIT.TEXT  changed to FOATISIT.FA
BUGS:WHOISIT.TEXT  changed to FOOISIT.FA
BUGS:WHYISIT.TEXT  changed to FOYISIT.FA
```

Here is another example using the original files shown above on the `BUGS:` volume. Specifying `BUGS:WH=.TEXT, =` results in:

```
BUGS:WHATISIT.TEXT changed to  ATISIT
BUGS:WHOISIT.TEXT  changed to OISIT
BUGS:WHYISIT.TEXT  changed to YISIT
```

You may wish to create some empty files using the Make command and experiment with them before using wildcards extensively. Until you get used to them, the effects of wildcards are not always obvious.

Note

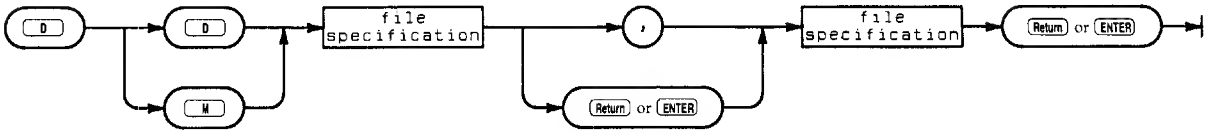
Using the Change command to "change" a file name to the same name results in the file being removed.

Note

The Change command does **not** change the file type.

Duplicate

The Duplicate link command establishes a new pointer to a file (SRM only).



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)

Semantics

The Duplicate link prompt:

```
Duplicate link (valid only for SRM type units)
Duplicate or Move ? (D/M)
```

Do you want the original pointer to the file removed after the duplicate link is established? If you do, type M — if not, type D .

The next prompt:

```
Dup_link what file ?
```

Type the SRM file specification (including the password if the CREATELINK capability has one).

```
Dup_link to what ?
```

Type the new file specification. Wildcards can be used in the specification. This puts a link to the file in a second directory. If the Move option was requested, the original link is then removed.

If the file is referenced from two or more directories, the file is physically removed from the disc only when all links to the file have been removed.

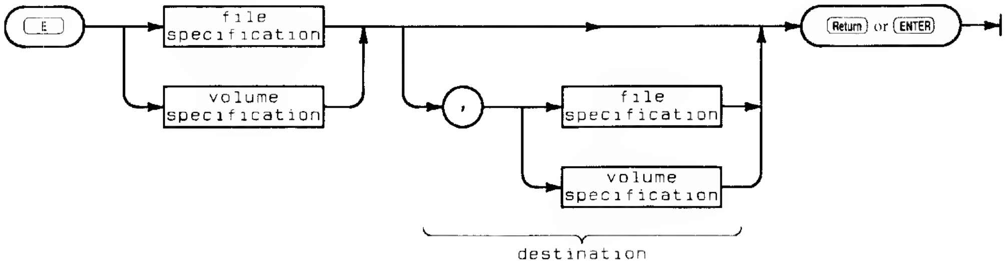
You should be aware that new CODE files generated by the Compiler, Assembler and Librarian to replace older versions are not written in the same space (unless Overwritten). If several directories have duplicate links to the same CODE file and the CODE file is recompiled, only one directory has an accurate link to the new CODE file. Other users must use the Duplicate link command to become linked to the new CODE file.

Note

Using the Duplicate command to "duplicate link" a file to the same file results in the file being removed.

Extended directory

The Extended directory command lists the directory of a blocked volume or a set of files in the volume.



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)
volume specification	literal	(See the beginning of this section)

Semantics

The Extended directory prompt:

```
List_ext what directory ?
```

The Extended directory command requires a legal volume ID or a file specification. Results can be listed to the PRINTER: or to a text file if specified and separated from the first specification by a comma. If no destination is specified the listing defaults to the CONSOLE. Wildcards are available to identify subsets of files on the volume.

In the listing, the name of the volume is displayed in the upper left-hand corner. To the right, the directory type is displayed. Pascal discs have Level 1 directories. Level 1 directories contain the creation date and volume size information. Level 0 directories (created on other systems) do not. Your directory listing should display the date the directory was created and the date it was changed as system volume, the size of the storage blocks, and whether the listing is in storage order or alphabetical order. The size of blocks on LIF volumes is 256 bytes. The size of blocks on WS1.0 volumes is 512 bytes. The Shared Resource Management system uses single byte "blocks".

To have directories listed in alphabetical order, include [*] after the directory name. For example:

```
MYDIR:[*]
```

The column entries for each file include: file name, number of blocks used for storage, the file size in bytes, the number of the block where the file starts, the date the file was changed, the type as recognized by the file system, the type-code used by the directory system, SRM access information, the date the file was created and two extension fields.

The SRM access information column comes under the heading "directory info". It contains codes which show the public access rights:

M	Manager
R	Read
W	Write
S	Search
P	Purgelink
C	Createlink

And the current file status:

```
CLOSED
SHARED
EXCLUSIVE
CORRUPT
```

CLOSED, SHARED and EXCLUSIVE are file status that are associated with SRM systems and are explained in detail in Chapter 2. If a file is ever marked CORRUPT, your Shared Resource Manager has a problem. Stop your operation and notify the person responsible for your SRM. He should restore the SRM to a usable state.)

The last two lines display additional directory information including how many more entries can fit in the directory. The size of a directory is specified when the disc is initialized.

The results can be listed to a printer or a file if you so specify. The destination of the listing is separated from the volume ID or file specification being listed by a comma and, if no destination is specified, the listing defaults to the screen. Wildcards are available to specify groups or subsets of files on a mass storage medium.

For example, assuming that SYSVOL: (the system volume) is in the disc drive associated with logical unit #3, a listing of all the CODE files on that volume could be sent to the printer by specifying any of the following in response to the Extended directory prompt:

`#3:=CODE,#6:` specifies volume residing in unit #3; listing to logical unit #6: (the PRINTER: volume)

`*=CODE,PRINTER:` specifies system volume; listing to the PRINTER. Without the colon, the listing would be sent to a DATA file named "PRINTER" on the default volume.

`SYSVOL:=CODE,#6` specifies SYSVOL: volume; listing to unit #6.

In all cases the "=CODE" string refers to all files whose names end in CODE on the specified volume and the listing is sent to the printer.

Listings can also be sent to a file. Use a destination parameter after the the source parameter (separated by a ",") as in the above PRINTER: example. Give a complete file specification. Use the appropriate suffix in the file name. Otherwise, a file of type DATA is produced. For example:

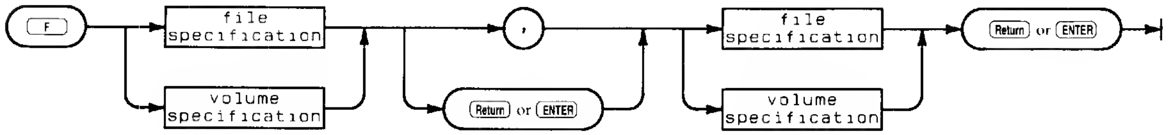
```
List what directory ? #3:,SYSVOL:LIST.TEXT
```

or

```
List what directory ? #3:,SYSVOL:LIST.ASC
```

Filecopy

The Filecopy command copies a specified file, set of files or volume to the specified destination.



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)
volume specification	literal	(See the beginning of this section)

Semantics

The Filecopy prompt:

```
Filecopy what file ?
```

The Filecopy command is initiated by pressing (F) and requires two specifications — a source and a destination separated by either a comma (,) or (Return) or (ENTER). The source volume must be on-line. The destination volume does not have to be on-line.

Wildcards may be used to specify sets of files. If the equals (=) wildcard is used, the copy is not confirmed before taking place. Also, note that if the equals wildcard is used alone (i.e., without any qualifying strings) then the Filer copies every file on the specified volume. If the question mark wildcard is used, you are asked to verify the transfer of each file meeting the wildcard specification before the Filecopy takes place. Thus, using the wildcard allows you more flexibility and control over the process.

The dollar sign character ($) may be used in the destination specification to indicate that the file(s) will have the same name (or names) as the source file(s). For example, assuming that there are a number of TEXT files on the volume TRIG: and that a second volume named MATH: exists,

```
TRIG:=TEXT,MATH:$
```

This results in all the files on the TRIG: volume whose file names end with the string TEXT being copied to the volume MATH: and given the same name as they have on the TRIG: volume.

Be sure to use either a file name or the $ character when specifying a destination volume. If, in the example above, the destination volume was specified as MATH: instead of MATH:$, the Filer would respond:

```
Destroy directory of MATH: ?
```

If you respond with (Y), the directory of that volume gets overwritten. Pressing (N) aborts the Filecopy command and returns the Filer prompt.

On a system with a single disc drive, the Filecopy command proceeds by reading the specified file or files into memory, prompting you to remove that volume and insert the destination volume, and then writing the file(s) in memory to the destination volume. Depending on the amount of memory in your computer and the amount of material being copied, you may have to swap discs more than once.

Note

When using the Filecopy command with a single disc drive, wait for the Filer's prompt before removing the source volume and replacing it with the destination volume. Failure to follow this guideline may result in the loss of information from the source volume.

A size specification may be used in the destination description. For example, specifying:

```
SYSVOL:FILE,OTHERVOL:FILE[35]
```

would result in the file being written to the first available area on OTHERVOL: that was at least 35 blocks in size.

To make a back-up copy of an entire volume, use the Filecopy command. Simply type in the source volume ID and the destination volume ID. The destination volume must be initialized but does not have to be Zeroed (the directory gets copied from the source volume). The Filer will ask you if you want the directory destroyed. A volume-to-volume copy makes an **exact** copy of the source volume on the destination volume.

Note that having two volumes with the same name on-line at one time is **not** advised. The Filer looks for volumes according to their volume names and may not be able to distinguish one from the other. Thus, the Filer may perform an action on one volume when you wanted the operation to affect the other volume. The Filer warns you whenever this condition exists. If you get a warning, either remove one of the volumes or use the Filer's Change command to change the name of one of the volumes.

You can copy files on one volume to a volume of a different size but you should not use volume IDs alone to do this. If the source volume is larger than the destination volume, the Filer refuses to execute the Filecopy. If the source is smaller than the destination, the destination volume ends up the same size as the source when the operation is through so you lose storage space. Remember? It makes an exact duplicate of the source.

The best way to handle copies between different size volumes is to use one of the wildcards. Use the equals wildcard (=) if the destination is larger than the source and the question mark wildcard (?) if the destination is smaller than the source. In the latter case you have to be selective in your copies since there is not enough space for all of the files.

When the Filecopy command has finished its task, the screen displays what file(s) or volume has been copied and the Filer prompt appears. The Filecopy command can be aborted before all specifications are complete by pressing Return or ENTER in response to the prompt.

In cases where the destination volume already contains a file with the same name as the file being copied, this prompt is displayed:

```
ANYVOL:XFILE
exists ... Remove, Overwrite, Neither ? (R/O/N)
```

You have the options:

- **Remove:** remove the file before proceeding with the copy operation.
- **Overwrite:** replace the contents of the old file with the new information. The Overwrite option cannot be used to change the type of a file on SRM. Attempting to do so will result in the file contents being inconsistent with the file type.
- **Neither:** cancel the operation.

The Overwrite option allows you to put a file in the same starting location as the original. This is important to SRM users when duplicate links exist to a file. All links and passwords to the file are accurate when a file is updated because it is put in the same logical location. If you chose the Remove option, the original file would not actually be removed; only your link to it is removed. The other users are still linked to the original file.

Note

Using the Filecopy command to "copy" a file name to the same name on the same volume results in the file being removed.

Note

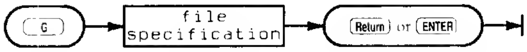
The Filecopy command does **not** change the file type.

Note

Overwrite of a file of type SYSTM is not recommended because the start execution address cannot be changed in an existing SYSTM file.

Get

The Get command associates a specified file as the current workfile.



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)

Semantics

The Get prompt:

```
Get what file ?
```

The Get command is initiated by pressing G and prompts you for a file specification. If a workfile currently exists when the Get command is executed, you are asked if you want to release that file before being allowed to specify a new workfile. Upon receiving the specification, the Filer finds the file (or files) and associates that name with the current workfile. Subsequent operations on the workfile use the specified name. The workfile is generally named *WORK.TEXT and/or *WORK.CODE.

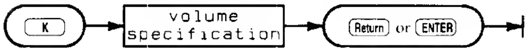
The Get operation assumes that the text version of the specified file has a .TEXT suffix. If the text version is ASCII, you must include the .ASC suffix. If the text version is DATA, you must include a "." at the end of the file name (to prevent the appending of the .TEXT suffix).

The operating system notes that either a text or code or both versions of the workfile exist. Workfiles can only be of type TEXT/ASCII/DATA or of type CODE. If both text and code versions of the specified file exist, both are associated with the workfile; if only one exists, the association is made with that file. The Filer reports one of three things: either a text or code file has been loaded, both have been loaded or the file cannot be found on the specified volume.

The Filer is not the only Pascal subsystem where a workfile can be created. The Compiler, Assembler and Editor subsystems also create workfiles. Once a workfile exists, it is treated as the default file in many of the subsystems. A workfile is "released" by the Filer's New command.

Krunch

The Krunch command moves all files on a block structured volume so that all the unused storage space is at the end of the volume.



Item	Description/Default	Range Restrictions
volume specification	literal	(See the beginning of this section)

Semantics

The Krunch prompt:

```
Crunch what directory ?
```

If there is the slightest question about the reliability of the medium you are using (because of excessive wear or damage), use the Bad sector command to do a scan of the sector on the volume **before** initiating Krunch. If a bad sector is found, use the Filer's Make command to make a file of type .BAD over the bad sectors. Krunch does not move files of type .BAD. Moving files onto an unreliable area of storage is a good way to lose a file.

The Krunch command is initiated by pressing K and it prompts you for a volume ID. After you respond with a legal volume ID of an on-line block structured volume, it prompts:

```
Crunch directory MKWORK: ? (Y/N)
```

Where `MKWORK:` is whatever volume you specified. Typing Y for Yes lets the command continue; N for No returns the Filer prompt. The Krunch command executes a sensitive operation -- that of moving all the files forward on the disc by reading the files into memory and then writing them back out on the disc in such a manner so as to make all the unused space on the volume contiguous at the end of the disc.

Note

UNDER NO CIRCUMSTANCES SHOULD YOU ATTEMPT TO INTERRUPT THE KRUNCH OPERATION ONCE IT HAS BEGUN. You are risking your directory and thus, all the information contained on that medium if you do so. Do not touch the power switch, the door on the disc drive or attempt to use the keyboard while a Krunch is occurring.

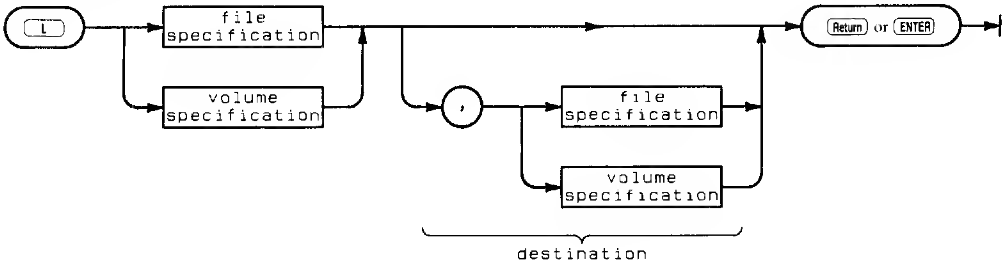
This process becomes necessary when, after repetitive reading and writing to the disc, the available storage space becomes highly fragmented. The situation can exist where you have 100 blocks available on the disc but because they are all in 10 or 15 block chunks, there is not enough contiguous storage space for the system to write a 20 block file to the disc.

The Krunch command is extremely useful and using it should not worry you. However, because it alters the directory (which maps where the information on the disc resides), it is one of the quickest ways to wipe out a volume. The precautions outlined above should help you avoid any problems while using the command.

The Krunch command does nothing on SRM units.

List directory

The List directory command lists directory information about a block structured volume or one of its subsets.



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)
volume specification	literal	(See the beginning of this section)

Semantics

The List directory prompt:

```
List what directory ?
```

The List directory command requires a legal volume or file specification. Results can be listed to the PRINTER: or to a text file if specified and separated from the first specification by a comma. If no destination is specified the listing defaults to the CONSOLE. Wildcards are available to identify subsets of files on the volume.

In the listing, the name of the volume is displayed in the upper left-hand corner. To the right, the directory type is displayed. Pascal discs have Level 1 directories. Level 1 directories contain directory-create and volume size information. Level 0 directories (created on other systems) do not. Your directory listing should display the date the directory was created and the date it was changed as system volume, the size of the storage blocks, and whether the listing is in storage order or alphabetical order. The size of blocks on LIF volumes is 256 bytes. The size of blocks on WS1.0 volumes is 512 bytes. The Shared Resource Management system uses single byte "blocks".

To have directories listed in alphabetical order, include [*] after the directory name. For example:

```
MYDIR:[*]
```

The column entries for each file include: file name, number of blocks used for storage, the file size in bytes, and the date the file was created or changed.

The last two lines display additional directory information including how many more entries can fit in the directory. The size of a directory is specified when the disc is initialized. You need one 256 byte block for each eight directory entries.

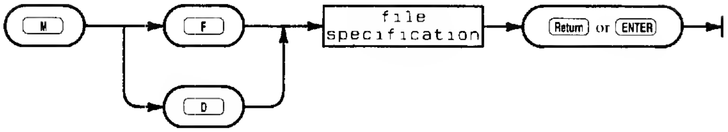
For example, initiating the command by pressing (L), specifying `ACCESS:` and pressing (Return) or (ENTER) results in the following listing appearing on the screen:

```
ACCESS:            Directory type= LIF level 1
created 20-Sep-82 13.57.17 block size=256
 Storage order
...file name....    # blks    # bytes  last chng

FILER                   218      55808 20-Sep-82
EDITOR                  224      57344 20-Sep-82
LIBRARIAN               202      51712 20-Sep-82
MEDIAINIT.CODE          132      33792 20-Sep-82
TAPEBKUP.CODE            54      13824 20-Sep-82
FILES shown=5 allocated=5 unallocated=3
BLOCKS (256 bytes) used=830 unused=223 largest space=223
```

The Extended Directory command gives more information about the files and unused areas on the volume.

Make

The Make command creates files and directories.



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)

Semantics

The Make prompt:

```
Make File or Directory ? (F/D)
```

The Make command is useful primarily in two ways. Files can be made when you need to reserve physical space on a disc, and directories can be made on an SRM system.

The Make command is **not** required to create files to be used by the various Pascal subsystems. It reserves space only; it in no way initializes or changes the contents of the space. In the Pascal System, each subsystem lets you either create or specify any files you need. Users of HP BASIC may quite naturally think that the same function is served by this command as the CREATE command in BASIC (where you must create a file before using it). Thus the distinction between these similar sounding commands is drawn here.

The Make command requires at least a file specification (which includes a volume ID by definition) and accepts an optional size specification. If the (positive integer) size is given, it must follow the file specification on the same line and be enclosed in square brackets. The Filer then creates a file with the specified name and of the specified size on the first area of the volume that has a large enough area of contiguous storage space to meet the size requirements.

When using a size specification to make a file, you must be aware that the size is specified in "number of blocks". The size of all "Make" blocks is 512 bytes — regardless of the directory type. A LIF directory considers a 256 byte sector to be a block. The WS1.0 directory considers a block to be 512 bytes. So if you make a file on a LIF volume and specify 500 blocks, it will show up in the directory as 1000 blocks.

For example, assume that there is a volume named MKWORK: on-line that has at least 22 blocks of contiguous and unused space available. Press M to initiate Make, specifying:

```
MKWORK:DUX.TEXT[22]     Return or ENTER
```

This results in a file named DUX.TEXT being created on the first available area with 22 blocks of the volume MKWORK: and the Filer reporting the following:

```
MKWORK:DUX.TEXT made
```

A subsequent listing of the directory (using the List directory or Extended directory commands) will show a file of the same name with a 22 block size (on WS1.0 directories).

The size specification may be omitted in which case the Filer creates the specified file using the largest unused area on the disc (i.e., the largest contiguous storage space on the disc will be allocated to the file). It is recommended that you specify the size you want the file to be.

There are two special cases of size specification worth knowing about. The first is the number zero enclosed in brackets `[0]` which is the same as omitting the size specification altogether — the Filer uses the largest space available. The second case is the asterisk character enclosed in brackets `[*]` which tells the Filer to make the file's size either the second largest area on the disc or half of the largest area, whichever is greater.

The Make command is useful if you must rebuild a file that was lost on a disc.

1. You must know its size and where it was located.
2. Then make TEMP files (e.g., TEMP1, TEMP2, etc.) over all the unused spaces on the disc that are as large or larger than the file you'll be making.
3. Then make a file of the proper type over the lost file to recover it.
4. Finally, use the Filer's Remove command to remove all the TEMP files.

An Extended directory listing can help you determine the location and size of unused areas on the disc.

The above technique will not recapture lost files on SRM systems. However, the Make command is used to create directories on an SRM system. For example:

```
Make File or Directory ? (F/D)
```

Answer the first question by typing D and specify where you want the directory located and what is its name. The directory path tells where you want it and its name is the name on the end of the path. For example, if you had an SRM directory:

```
#5:USERS/JOE/PROJECT1
```

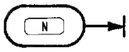
If you wanted to create a directory for Project 1's DATA files, you should type:

```
#5:USERS/JOE/PROJECT1/DATA
```

The DATA directory is created in the PROJECT1 directory.

New

The New command releases or clears the workfile area.



Semantics

The New command requires no specifications. Upon pressing N to initiate the command, it clears the workfile unless the workfile has been updated since the last Save command. If this is the case, the prompt appears:

```
Throw away current workfile? (Y/N)
```

Responding by pressing N for No allows you to use the Save command to write the file to a volume; Y for Yes clears the current workfile area. When the Filer executes the New command, it will respond with:

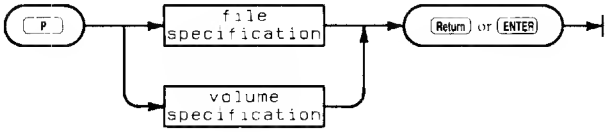
```
Workfile cleared
```

You can check the status of the workfile before using New with the What command. The What command gives you the name and status (saved or not) of the current workfile.

Do not confuse the Filer's New command with the New system volume command at the Main Command Level — the two commands are different and perform separate functions.

Prefix

The Prefix command changes the default volume to the one specified.



Item	Description/Default	Range Restrictions
file specification	literal	for SRM only (See the beginning of this section)
volume specification	literal	(See the beginning of this section)

Semantics

The Prefix prompt:

```
Prefix to what directory ?
```

The Prefix command is initiated by pressing (P) and requires a volume ID. The command allows you to specify a new default volume — the one where the Filer searches for file specifications when a volume name is not specified. The volume must be block structured (one used for mass storage) but does not have to be on-line. The current prefix (i.e., default) volume can be obtained by responding to the Prefix prompt with a colon (:). (The Volumes command may also be used).

When the command executes, the screen displays the message:

```
Prefix is MKWORK:
```

Where MKWORK: is the name of the current default prefix. The Prefix command saves keystrokes if you are doing a lot of file accessing on a particular volume.

Filer commands which request a volume ID may be answered with the colon character (:) which specifies the current default volume.

It is possible to set the default prefix to a flexible disc drive, regardless of the volume inside. This is done by typing the following while the drive door is open.

#3: (Return) or (ENTER)

The prefix command is used to set up a working directory on a Shared Resource Management system as well. If you had an SRM file named:

```
#5:USERS/JOE/PROJECT1/PROGRAMS/FILE
```

Initiate the Prefix command as usual and specify:

```
#5:USERS/JOE/PROJECT1/PROGRAMS
```

This sets the SRM volume as the default volume (with volume name of PROGRAMS) and USERS/JOE/PROJECT1/PROGRAMS as the working directory on the SRM. Now you can specify the file with:

```
FILE
```

If you were to use the Prefix command again to set the default prefix to another volume (not on the same unit), the working directory and volume name for the unit remain PROGRAMS. You need only specify:

```
#5:FILE
```

or

```
PROGRAMS:FILE
```

Either will get the same file.

Note

Do not use the Prefix command on unit #45. This is the system volume and should not be altered.

It is possible to change the working directory on an SRM unit without changing the default volume. Use the Filer's Unit directory command. Press (U) and then give the directory name that you wish to become the working directory. If the new directory is in the existing working directory, just type the new directory name. If it is not, type the whole directory path as shown above in the Prefix example.

Quit

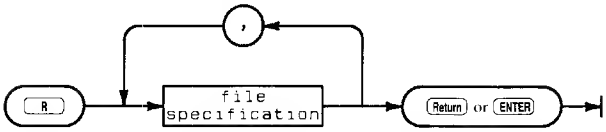
The Quit command exits the Filer subsystem and returns control to the Main Command Level.

Semantics

The Quit command has no parameters and no specifications of any type are needed. Pressing Q exits you from the Filer and the Main Command Prompt is displayed on the screen.

Remove

The Remove command purges specified files from the directory.



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)

Semantics

The Remove prompt:

```
Remove what file ?
```

The Remove command is initiated by pressing R and requires a file specification. The command removes the specified file from the directory, updates the directory and reports the action it has performed. Wildcards may be used to specify a subset of files to be removed. If the equals wildcard (=) is used in the file specification, the Filer reports the specified file or files and then prompts:

```
Proceed with remove ? (Y/N)
```

This is the last chance you have to change your mind about the removal. Pressing N for No aborts the operation and no files are removed. Pressing Y for Yes removes those files meeting the wildcard specification from the directory. The process is not always reversible. However, the Make command can sometimes be used to recover a removed file.

Note

The Filer considers the file specification = to specify ALL the files on the default volume and MKWORK: = to specify ALL the files on the MKWORK: volume. If you use the wildcard in this form and respond to the Filer's prompt (`Proceed with remove ? (Y/N)`) with a Y for Yes, every file on the directory of the specified volume is removed. Responding with a N for No aborts the operation. Wildcards can be hazardous to your files — watch the prompts.

Specifying a single file (of an on-line volume of course) in response to the Remove prompt results in the removal of that file from the directory and a report that the file has been removed. Once the Return or ENTER key is pressed following the file specification (unless wildcards are used), that file is gone.

While the use of the equals wildcard (=) results in being prompted for whether or not you want the directory updated, the question mark wildcard (?) acts slightly differently. It allows you to be more selective in your removal. Given the volume PROCESS: containing the files:

```
NOVMEMO.TEXT
MARKLTR.TEXT
PARSER.TEXT
PARSER.CODE
GARBAGE.TEXT
```

The specification PROCESS:?TEXT in response to the Remove prompt results in the screen clearing and the following message appearing.

```
Remove NOVMEMO.TEXT ? (Y/N)
```

Answering with either a Y or N results in the next prompt appearing below the first:

```
Remove MARKLTR.TEXT ? (Y/N)
```

The process continues until you have been prompted for all the TEXT files on the `PROCESS:` volume and then the final prompt appears:

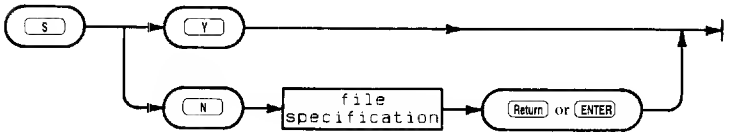
```
Proceed with remove ? (Y/N)
```

You may be respond with either a Y (for Yes) or N (for No). The files are not actually removed until this prompt is answered with a Y. The ? wildcard thus allows you to be both selective and relatively safe about your file removals.

The Remove operation treats SRM directories like files if they are empty. Remove is not allowed on non-empty SRM directories.

Save

The Save command saves the current workfile on the specified volume.



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)

Semantics

The Save command is initiated by pressing S and may or may not require a file specification. If the workfile was never updated, it is automatically saved with the original name.

If the workfile was previously named using the Save command, or originally obtained using the Get command, then the Filer prompts:

```
Save as PREVIOUS.TEXT ? (Y/N)
```

Where PREVIOUS.TEXT is the name previously associated with the workfile. Responding with a Y for Yes results in either a CODE or TEXT file (or both, depending on what is in the workfile) of that name being removed and replaced with the current workfile.

If the workfile is not named, or if you answer N, the Filer prompts:

```
Save as what file ?
```

When naming the file, the following conventions apply to the type of the file:

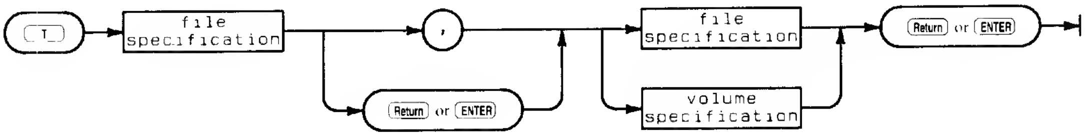
1. If a standard suffix is recognized, the workfile is either Filecopied, Translated or Changed (on the system volume) to the file name and type.
2. If no suffix is recognized, a .TEXT file is the default.
3. If no suffix, but a "." is found, the file type is DATA.
4. The .CODE file is created by removing the suffix (if there is one) and adding .CODE to the file name.

The Filer displays that the file is now saved.

To find out what the current name and state (saved or not) of the workfile is, use the What command.

Translate

The Translate command converts text files between the TEXT, ASCII, and DATA formats.



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)
volume specification	literal	(See the beginning of this section)

Semantics

The Translate prompt:

```
Translate what file ?
```

The Translate command is initiated by pressing T and requires two specifications — a source and a destination separated by either a comma (,) or a carriage return (press Return or ENTER). The source specification can be any block structured volume, any file or any group of files on a volume. The destination specified can be any of the above and may also be a non-block structured volume (i.e., the `PRINTER:` or `CONSOLE:`). Non-block structured volumes (like the PRINTER:) are assumed to be on-line.

Wildcards may be used to specify sets of files but if a wildcard is used in the source specification, either a wildcard or the `$` character (discussed below) must be included for the destination. If the equals wildcard (=) is used, the translate is not confirmed before taking place. Also, note that if the `=` wildcard is used alone (i.e., without any qualifying strings such as `LIBR`, `TEXT`, etc.) then the Filer Translates every file on the specified volume. If the question mark wildcard (?) is used, you are asked to verify the translate of each file meeting the wildcard specification before the Translate takes place. Thus, using the `?` wildcard allows you more flexibility and control over the process.

The dollar sign character ($) may be used in the destination specification to indicate that the file(s) will have the same name (or names) as the source file(s). For example, assuming that there are a number of TEXT files on the volume `TRIG:` and that a second volume named `MATH:` exists,

```
TRIG:=TEXT,MATH:$
```

This results in all the files on the `TRIG:` volume whose file names end with the string `TEXT` being translated to the volume `MATH:` and given the same name as they have on the `TRIG:` volume.

On a system with a single disc drive, the Translate command proceeds by reading the specified file or files into memory, prompting you to remove that volume and insert the destination volume, and then writing the file(s) in memory to the destination volume. Depending on the amount of memory in your computer and the amount of material being translated, you may have to swap discs more than once.

Note

When using the Translate command with a single disc drive, wait for the Filer's prompt before removing the source volume and replacing it with the destination volume. Failure to follow this guideline may result in the loss of information from the source volume.

The Translate command allows the translating of files or groups of files to non-block structured devices like the `PRINTER:` and `CONSOLE:`. Only text (TEXT, ASCII, DATA) files should be sent to printers since other files are not generally human readable.

When the Translate command has finished its task, the screen displays what file(s) have been translated and the Filer prompt appears. The Translate command can be aborted before all specifications are given by pressing (Return) or (ENTER).

In cases where the destination volume already contains a file with the same name as the file being Translated, this prompt is displayed:

```
ANYVOL:XFILE
exists ... Remove, Overwrite, Neither ? (R/O/N)
```

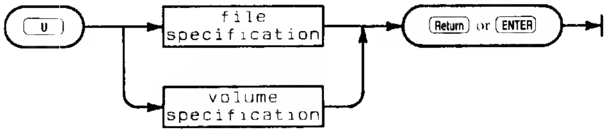
You have the options:

- **Remove:** remove the existing file before proceeding with the translation.
- **Overwrite:** replace the contents of the old file with the new information. The Overwrite option cannot be used to change the type of a file on SRM. Attempting to do so will result in the file contents being inconsistent with the file type.
- **Neither:** cancel the operation.

The Overwrite option allows you to put a file in the same starting location as the original. This is important to SRM users when duplicate links, passwords, etc. exist to a file. All links and passwords to the file are accurate when a file is updated because it is put in the same logical location. If you chose the Remove option, the original file would not actually be removed; only your link to it is removed. The other users are still linked to the original file.

Unit directory

The Unit directory command changes the volume name and working directory for an SRM unit.



Item	Description/Default	Range Restrictions
file specification	literal	(See the beginning of this section)

Semantics

The Unit directory prompt:

```
Set unit to what directory ?
```

The Unit command changes the working directory on SRM units. The working directory and the volume name for SRM units are the same. The Prefix command performs the same operation but sets the default volume to the SRM volume. The Unit command does not.

To specify the working directory, you must start either from the existing working directory or from the root directory.

To get to the root, an SRM volume identifier must be given or the default is assumed. Follow the volume ID with "/". This positions you in the root directory.

From the working directory, you can continue down the tree structure from directory to directory or you can go back up the structure one directory at a time using ".." for the parent of the current directory.

For example, if the present working directory for unit #5 is:

```
USERS/JOE/PROJECT1/PROGRAMS
```

and you wanted the new working directory to be:

```
USERS/JOE/PROJECT5/OOCUMENTS
```

you can specify it in one of the following ways.

```
#5:/USERS/JOE/PROJECT5/OOCUMENTS
```

or:

```
PROGRAMS:/USERS/JOE/PROJECT5/OOCUMENTS
```

or:

```
#5:../../PROJECT5/DOCUMENTS
```

or:

```
PROGRAMS:../../PROJECT5/OOCUMENTS
```

Volumes

The Volumes command lists the volumes currently on-line.

Semantics

The Volumes command requires no specifications. Upon pressing V it displays the following information about all on-line volumes currently associated with the Pascal System: the logical unit number associated with a volume, whether the volume is the system (boot) volume, a block structured volume or a non-block structured volume, the volume's name, and the current Prefix or default volume.

This is a typical display generated by the Volumes command:

```
Volumes on-line:
  1     CONSOLE:
  2     SYSTERM:
  3 #   MINI3:
  4 #   MINI4:
  5 #   MY_SRM:
  6     PRINTER:
 45 *   SYSTEM04:
Prefix is - MY_SRM:
```

The number on the far left is the logical unit number associated with the volume. The * character in the second column indicates the system volume which is always block structured. The # character indicates all other block structured volumes currently on-line. The remaining volumes (shown with no character in the second column) are non-block structured. The last line of the display shows the current default volume. It is where the system looks for a file when no volume has been specified.

The above configuration shows two 3.5 or 5.25-inch flexible disc drives associated with units #3 and #4 and two SRM volumes associated with units #5 and #45. Respectively, they are the working volume and system volume.

What

The What command displays the name and state (saved or not) of the workfile.

Semantics

The What command is initiated by pressing W and requires no other input. The command shows the name of the current workfile or indicates that it is not associated with a file name. It also shows whether or not the workfile has been saved since the last update to the file. If no workfile exists, the Filer responds with:

```
No workfile
```

Suppose you had two files named INFRARED.TEXT and INFRARED.CODE on the default or prefix volume. Assume that you used the Filer's Get command and specified INFRARED to associate the files with the workfile. If you then edited the TEXT version of that file (using the Pascal Editor), returned to the Filer and executed the What command, the screen would display:

```
Workfile is INFRARED (not saved)
```

because the workfile was changed since the last time a Save command was executed.

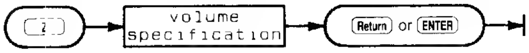
Saving the workfile does not change the fact that the workfile exists. It is still there. The New command is used to clear the workfile. Saving the workfile is not remembered between separate sessions of the Filer. If you Save the workfile during the current Filer session, a New command immediately clears the workfile. If you Save it, quit the Filer and then return to use the New command, the Filer will ask:

```
Throw away current workfile ? (Y/N)
```

even though you saved it during the previous Filer session and haven't updated it since.

Zero

The Zero command creates an empty directory on the specified volume. The Zero command is not allowed on SRM volumes. (See the Make command.)



Item	Description/Default	Range Restrictions
volume specification	literal	(See the beginning of this section)

Semantics

The Zero prompt:

```
Zero directory (NOT valid on SRM type units)
Zero what directory ?
```

The Zero command is initiated by pressing (Z) and requires the volume ID of a block structured volume. The volume must be formatted using the Pascal utility program `MEDIAINIT.CODE` supplied on the `ACCESS:` volume.

Since the Zero command creates an new empty directory on the volume, you will be prompted:

```
Destroy THISVOL: ? (Y/N)
```

Responding with a (N) for No aborts the command and returns the Filer prompt.

If you answer (Y), the next prompt is:

```
Number of directory entries (8) ?
```

The number in the parentheses is the number in the existing directory. Respond with (Return) or (ENTER) if that is the number you want. If there is no number in parentheses, (Return) or (ENTER) causes the default number for that directory type (80 for LIF; 77 for WS1.0) to be put on the disc.

The next prompt is:

```
Number of bytes (270336) ?
```

It is asking for the logical size of the disc (the extent to be managed by the directory). The number in the parenthesis is the number in the existing directory or the default for that disc. Press Return or ENTER to use the displayed number.

The next prompt is:

```
New volume name ?
```

The Filer is asking for a legal volume name. Volume name formats vary with different directory structures. LIF directories allow up to six characters with upper and lower case characters being distinct. WS1.0 directories allow up to seven characters and all are made upper case.

An answer of Return or ENTER aborts the Zero command.

After typing a volume name, the final prompt appears:

```
NEWSTUF: correct ?
```

Responding with N aborts the Zero command. Responding with Y results in the message:

```
NEWSTUF: zeroed
```

Where `NEWSTUF:` is the name of the new volume. The Filer prompt reappears when the operation is complete.

Note

Because the file system works with volume names, a LIF volume whose name is all blanks (ASCII spaces) will not be recognized as a valid volume.

Notes

Pascal Compiler

Chapter 5

Introduction

The Compiler supplied with your system accepts the Pascal language specified by a Hewlett-Packard Company standard. HP Pascal is designed to be a superset of the ISO Pascal Standard. This language implementation is described in the *HP Pascal Language Reference for Series 200 Computers.*

In addition, Compiler options can be used to enable or disable certain language features. Under control of these options, the Compiler can also process

- Programs restricted to the ISO definition
- Almost all the features of UCSD Pascal[1]
- Certain language extensions needed for system programming applications

UCSD Pascal and system programming extensions are described in this chapter and in the *HP Pascal Language Reference for Series 200 Computers.*

HP Pascal is intended to be as portable as possible across the many computers and operating systems used in HP. Use of the UCSD or system programming extensions may produce programs which are hard to transport. Whenever processing of non-standard language features is enabled, the Compiler displays a warning on the CRT and at the end of the compilation listing.

The Compiler is a one-pass design. It takes as input one or more source text files built by the Editor, and generates an output file of relocatable MC68000 object code, ready to be linked and run. Relocation and linking happen automatically when you execute the Run command — normally there is no explicit link step.

Compilation speed depends on the mass storage device where the source and object code reside. With floppy discs, about 1600 lines per minute is typical. If the files are memory-resident, the rate is around 4000 lpm. The Compiler's speed contributes significantly to the interactive and crisp feeling of the Pascal environment.

The Compiler, supported by other subsystems, provides complete facilities for the creation, maintenance and use of software libraries. Modules of Pascal code can be compiled, stored in the system library, and automatically accessed by any program which needs them. Compiled modules carry along a detailed specification of their interface to any program which uses them.

[1] "UCSD Pascal" is a trademark of the Regents of the University of California.

Steps In Program Development

This section will teach you by example the steps required to compile and run a simple program. You need to know how to use the Editor before you can proceed with this material. We begin at the Command level of the system, with no workfile present.

Prepare the Source Program

First we need a program to compile. Enter the following program using the Editor. The Compiler isn't particular about margins, so you can adjust the program to the left margin as you type. Try to preserve the indentation, to keep the program easily readable by mortals.

Notice that the word "end" is intentionally misspelled at the bottom of the program. Type it just as shown, so you can see how errors are handled.

When you leave the Editor (Quit command), you should specify that the output is to be Written to the file "HOWDY". Don't make a workfile (don't use the Update option).

```
program howdy (input,output);
type
  color = (red,orange,yellow,green);
var
  hue: color;
  i: integer;

  procedure show (c:color);
  begin
    writeln(output,'Howdy!   ',c);
    i := i+1;
  end;

begin
  writeln(output);  i := 0;
  for hue := red to green do
    show(hue);
emd.
```

At this point, if you use the Filer to examine the directory of your default volume, you'll see the file "HOWDY.TEXT" .

Invoke the Compiler

The Compiler is invoked by typing the C key when the system is at the Main Command Level. At the time you booted up, the system looked for the Compiler on all the mass storage volumes which were on-line. If the Compiler was found at that time, it is expected to still be in the same volume whenever it's needed. If the Compiler wasn't found, the system will try to run CMP:COMPILER (or the file specified with the last What command).

So if you press C and the system responds that it can't load the Compiler, you must **first** put the CMP: disc in a drive, **then** press C again.

It takes a few seconds for the Compiler to load from a floppy disc. Then it will ask you,

```
Compile what text ?
```

If you had to swap discs, you should remove the CMP: disc and put back the default volume. Now respond:

HOWDY Return or ENTER

The Compiler automatically appends the ".TEXT" suffix to the name you give; you need not do so yourself. Next you are asked,

```
Printer listing (l/y/n/e) ?
```

If you have no printer, you **must** answer N for no listing or L for a listing file. If you've got a printer, the Y response gets you a complete listing. Answering E will get you a listing only of any errors which are detected. For the moment, let's answer N and get no listing. Finally the Compiler asks:

```
Output file (default is "HOWDY.CODE") ?
```

Respond to this by pressing Return or ENTER to accept the default.

As the Compiler runs, you can observe its progress through the source program. Each dot displayed represents five lines of the source text which have been scanned. Whenever the body (the "begin") of a new procedure is reached, that procedure's name is displayed on the screen along with an estimate (in square brackets) of how much memory is still available for the Compiler to use. The Compiler reads through an entire procedure body before generating any code; if you write very large procedures, you may notice the stream of dots hesitating momentarily at the ends of some of them.

When the misspelled word "emd" is encountered, the Compiler will beep and display the offending line. You now have three options: press the space bar to continue compiling, hold down SHIFT and press Select (EXECUTE) to terminate the compilation, or enter the Editor to fix the mistake. You should select "Edit" by pressing E.

Note

The Editor must be Permanently loaded, or the volume containing the Editor must be on-line to use the E option when exiting the Compiler.

Handling Syntax Errors

If the Compiler is printing a listing, it will report errors on the printout, rather than interactively, giving you an opportunity to edit the program. In this case, you must call the Editor yourself after the compilation is finished.

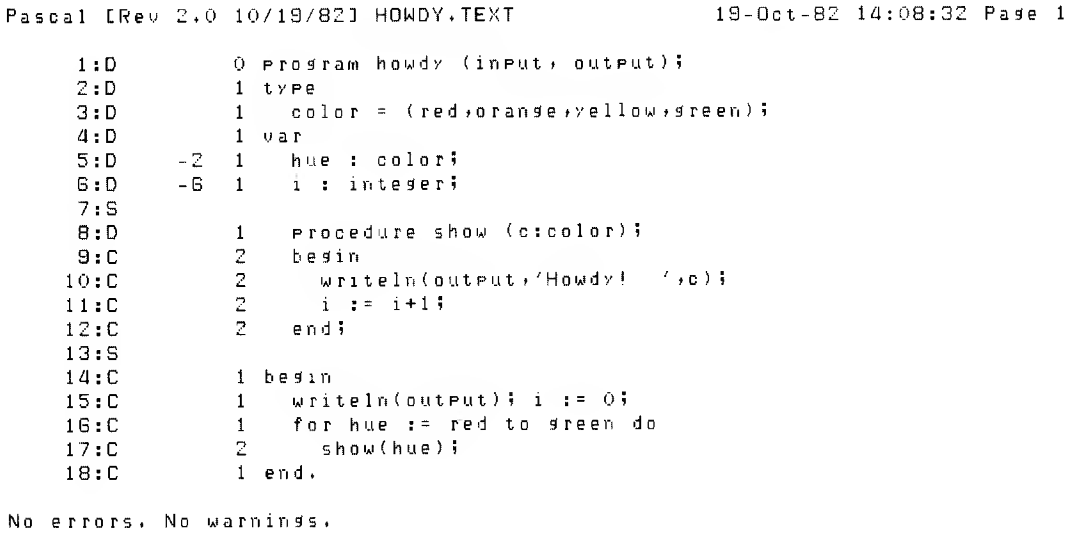
When the Compiler points out a syntax error, the place it indicates is not necessarily the place where the error occurred; rather, you are shown where the error was first **recognized**. An easy way to get extreme examples of this is to accidentally have unbalanced "begin" and "end" pairs in a deeply nested program. The imbalance may be **syntactically** (though not visually) undetectable until much later in the program. Compilers don't see what you mean, only what you write

The error message may not seem reasonable to you. For instance, your misspelled "end" looks to the Compiler like an undeclared identifier which may be the beginning of an assignment statement. The Compiler sees no similarity between "end" and "emd".

When an error is detected, the Compiler tries to recover by making an assumption about what you meant. Frequently the assumption is wrong, which leads to further errors being reported in the vicinity of the first one. Sometimes the Compiler will try to recover by skipping text until it sees a keyword or other symbol it recognizes.

Back to the example: you elected to edit the program, so the Compiler terminated and the Editor is now invoked. The file containing the offending line is automatically brought in, and the cursor is placed where the error was reported. Simply fix the misspelling and quit the Editor, using the Save option to rewrite the corrected file under its original name, "HOWDY".

Repeat the steps above to compile HOWDY again. If you have a printer, this time you should ask for a listing. If there are no other accidental errors, the compilation will succeed this time. Your printout should look like this:



```
Pascal [Rev 2.0 10/19/82] HOWDY.TEXT                19-Oct-82 14:08:32 Page 1

       1:D          0 program howdy (input, output);
       2:D          1 type
       3:D          1   color = (red,orange,yellow,green);
       4:D          1 var
       5:D      -2  1   hue : color;
       6:D      -6  1   i : integer;
       7:S
       8:D          1   procedure show (c:color);
       9:C          2   begin
      10:C          2     writeln(output,'Howdy!   ',c);
      11:C          2     i := i+1;
      12:C          2   end;
      13:S
      14:C          1 begin
      15:C          1   writeln(output); i := 0;
      16:C          1   for hue := red to green do
      17:C          2     show(hue);
      18:C          1 end.

No errors. No warnings.
```

Interpreting the Compilation Listing

The column of numbers at the left enumerates the lines. "D" next to the line number indicates the line is a declaration; "S" indicates the line was skipped altogether, either because it's blank, or because it is entirely within a comment. "C" indicates the line is part of the body of a Pascal block.

The two numbers, -2 and -6, provide information about where the variables "hue" and "color" will be stored in memory. More detailed information about this can be requested by the $TABLES$ Compiler option.

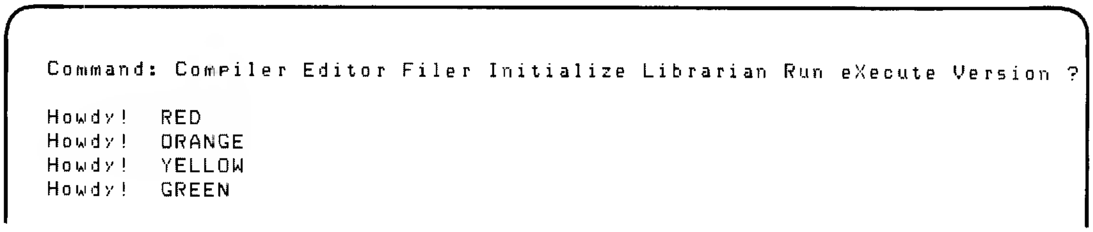
The column of numbers immediately to the left of the program text shows how deep structures in the program are nested. This can be very useful when begin's and end's get out of balance. The main program is at level 1, with procedures nesting successively deeper. The structural nesting of complex statements such as for-loops, if's and with's is also counted.

Running the Compiled Program

If you use the Filer to look at the directory of your default volume, you'll see that there are two HOWDY files now: HOWDY.TEXT and HOWDY.CODE . Press the R or RUN key. The operating system remembers the name of the most recently compiled file. You'll see the message,

```
Loading 'HOWDY.CODE'
```

The program runs, producing this display on the screen:



```
Command: Compiler Editor Filer Initialize Librarian Run eXecute Version ?

Howdy!  RED
Howdy!  ORANGE
Howdy!  YELLOW
Howdy!  GREEN
```

You can also run the program by using the eXecute command: press the Select (EXECUTE) or the X key. Then when asked

```
Execute what file ?
```

answer

```
HOWDY Return or ENTER
```

Try it now. Actually, you can eXecute any program, not just the one you most recently compiled. Also, if you use the Run command when you haven't compiled any program, the behavior is as if you used the eXecute command.

Using a Workfile

The Compiler's behavior depends somewhat on whether you are compiling a workfile, or some other source file. If you use a workfile, you are asked fewer questions by the Compiler and Editor; in fact, while the workfile is present you can't compile or edit any other file! This kind of abbreviated behavior may be a blessing or a curse, depending on your needs.

Workfiles are most useful when you're writing a small program and you're in a hurry. In that case you'll appreciate the convenient reduction in keystrokes needed to compile and run the program. On the other hand, experienced programmers developing complex systems with many source files almost never use workfiles.

Workfiles are **not** particularly useful unless the Editor has been permanently loaded, or the volume containing the editor is on-line. This is particularly important in systems which have no external mass storage.

There are two ways to tell the system to use a workfile. You can create one by using the Update option when quitting the Editor; a workfile made this way will always be called WORK.TEXT, and it will be stored in the system volume. Alternatively, you can designate some existing file as the workfile by using the Filer's Get command. The Update option and the Get command are explained in the Editor and Filer chapters of this manual.

Let's make a workfile of HOWDY using the Editor. Press the E key, and answer that you want to edit HOWDY. Immediately use the Quit command, and select the option to Update the workfile. This makes a **copy** of your original source file (but not of the code file). Note that the system volume must be on-line at this point, since that is where the workfile is kept.

Now press R. If the Compiler isn't on-line, you will need to insert your CMP: disc first. If you swapped disks, then after the Compiler is loaded it will say:

```
Mount *WORK.TEXT and press <space>
```

As you can see, the Compiler knows it's supposed to compile the workfile, and you must put your system volume back in the drive. If the Compiler was already on-line, only one question is asked after you press R:

```
Printer listing (l/y/n/e) ?
```

Probably you'll answer no. The program then compiles, producing WORK.CODE, and immediately runs.

To execute it again, just press R. It won't be recompiled unless you use change it with the Editor. If you aren't convinced it actually ran again (it happens pretty fast), press the space bar to clear the screen before running it again.

Debugging

The Debugger subsystem is described in detail in a later chapter of this manual. With Pascal 3.0, the Debugger is not automatically loaded at boot time. You will need to load it if you want to use it. See the Debugger chapter for loading instructions.

Modules

A **module** is a program fragment which can be compiled independently and later used to complete otherwise incomplete programs. For example, you might want to define a "complex number" data type and some relevant functions, then use those definitions in several programs. This section introduces the concepts and facilities you will need to define, debug and use module libraries.

Modules, like almost everything else in Pascal, must have all their relevant features and characteristics declared before use. Syntax diagrams detailing precisely the syntax of a module declaration can be found in the *HP Pascal Language Reference for Series 200 computers*; an informal presentation is more suitable for present purposes.

Module Structure

The four parts of a module are its heading, the import and export sections, and its implement part.

- The **heading** introduces the module and names it. The name is an ordinary Pascal identifier. Example:

  ```
  module complexmath;
  ```

- The **import** part names all other modules on which the present one depends. One module depends on another if the dependent module makes use of things exported from the imported one: calling procedures, assigning to exported variables, or declaring variables of an exported type. The names are separated by commas and the list ends with a semicolon:

  ```
  import complexmath,conversions;
  ```

  There is no import part if the module is independent of all others.

- The **export** part defines the constants, types, variables, procedures and functions which this module will supply to any program or module importing it. Constants, types and variables are declared just as in a program or procedure block. Procedures and functions are presented as headings without bodies.

  ```
  export
    const pi = 3.14159;
    type
      polar = record radius,theta: real end;
    var
      scalefactor: real;
      origin: polar;
    function makepolar (a: complex): polar;
    procedure setorigin (a: complex);
  ```

  The export part may make use of things in turn exported from other modules listed in the import part, such as the type "complex". Every module must have an export part.

- The **implement** part consists of the reserved word IMPLEMENT, followed by constant, type, variable, procedure and function declarations, followed by the word END. All the procedures and functions whose headings were in the export part must be present in their entirety in the implement part. The implement part may make use of things in turn exported from other modules listed in the import part.

  A module does not have to export procedures or functions, it may be used simply to create data types. In such a case there will be nothing between the words IMPLEMENT and END.

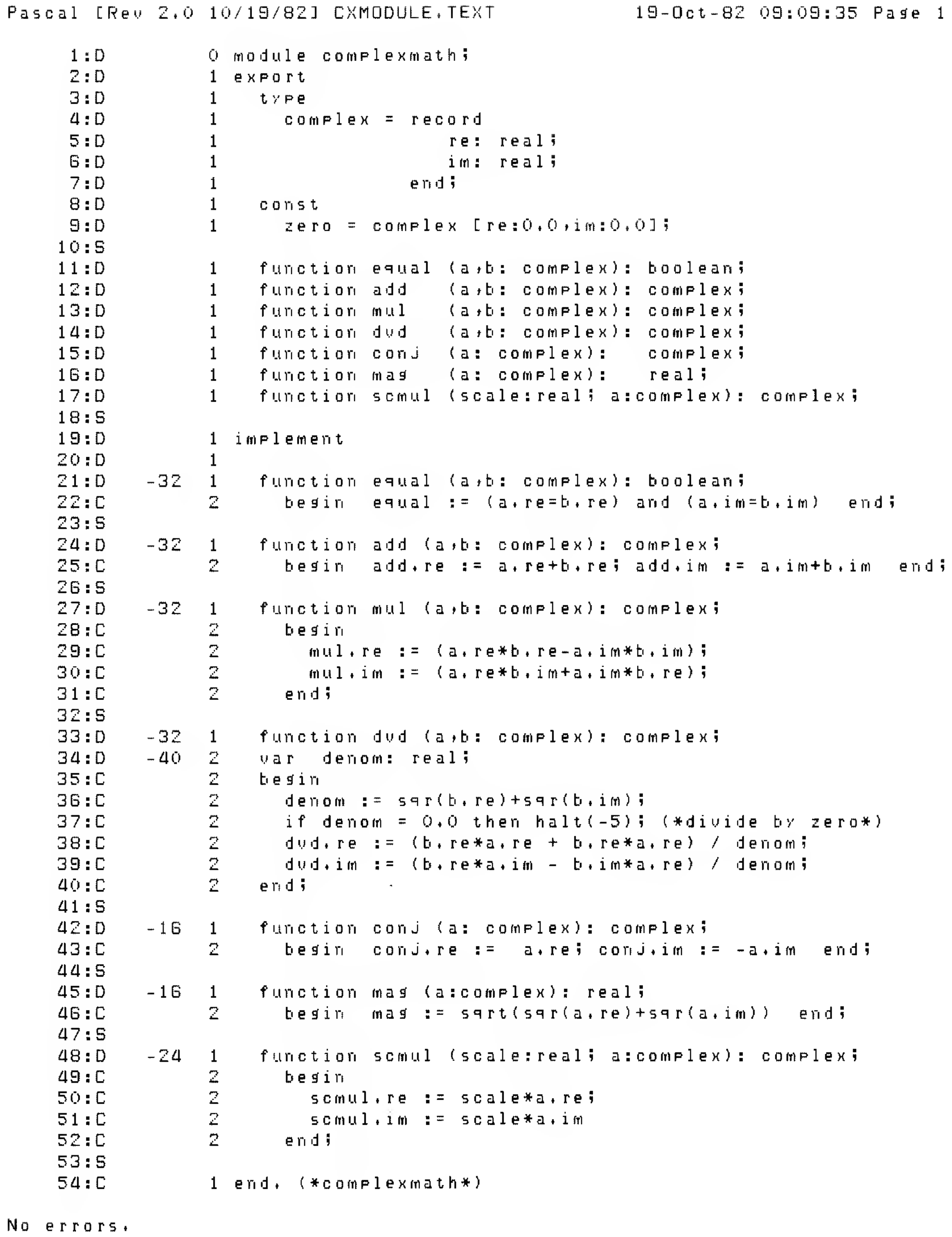
A complete module, "complexmath", is shown on the next page. It has no import part because it depends on no other modules. (The module is also on the DOC: disc; the source is called CXMODULE.TEXT.)

The import and export parts are said to define the module's interface to other modules or programs. This interface is public: the information it contains is available to any imported of the module.

The implement part is said to be "private", which means that everything between the words IMPLEMENT and END is hidden from importers. Anything declared here is unknown outside the module, except for procedures and functions whose headings were also included in the export part.

The private and public parts of the module are separated in this way so that its implement part can safely be changed without altering programs or other modules which import it. This independence of modules from programs is a key to developing software libraries. Another implication is that modules can only be dependent on other modules, not on programs. The reason is simply that there's no way to import a program into a module (since programs have no export declarations).

It was stated at the outset that a module is a "fragment" of a program. To be more precise, a module is a set of global (outer level) declarations which can be compiled once, then bound into a program by an IMPORT declaration in that program.



```
Pascal [Rev 2.0 10/19/82] CXMODULE.TEXT          19-Oct-82 09:09:35 Page 1

     1:D        0 module complexmath;
     2:D        1 export
     3:D        1   type
     4:D        1     complex = record
     5:D        1                  re: real;
     6:D        1                  im: real;
     7:D        1               end;
     8:D        1   const
     9:D        1     zero = complex [re:0.0,im:0.0];
    10:S
    11:D        1   function equal (a,b: complex): boolean;
    12:D        1   function add   (a,b: complex): complex;
    13:D        1   function mul   (a,b: complex): complex;
    14:D        1   function dvd   (a,b: complex): complex;
    15:D        1   function conj  (a: complex):   complex;
    16:D        1   function mag   (a: complex):   real;
    17:D        1   function scmul (scale:real; a:complex): complex;
    18:S
    19:D        1 implement
    20:D        1
    21:D   -32  1   function equal (a,b: complex): boolean;
    22:C        2     begin  equal := (a.re=b.re) and (a.im=b.im)  end;
    23:S
    24:D   -32  1   function add (a,b: complex): complex;
    25:C        2     begin  add.re := a.re+b.re; add.im := a.im+b.im  end;
    26:S
    27:D   -32  1   function mul (a,b: complex): complex;
    28:C        2     begin
    29:C        2       mul.re := (a.re*b.re-a.im*b.im);
    30:C        2       mul.im := (a.re*b.im+a.im*b.re);
    31:C        2     end;
    32:S
    33:D   -32  1   function dvd (a,b: complex): complex;
    34:D   -40  2   var  denom: real;
    35:C        2   begin
    36:C        2     denom := sqr(b.re)+sqr(b.im);
    37:C        2     if denom = 0.0 then halt(-5); (*divide by zero*)
    38:C        2     dvd.re := (b.re*a.re + b.re*a.re) / denom;
    39:C        2     dvd.im := (b.re*a.im - b.im*a.re) / denom;
    40:C        2   end;
    41:S
    42:D   -16  1   function conj (a: complex): complex;
    43:C        2     begin  conj.re :=  a.re; conj.im := -a.im  end;
    44:S
    45:D   -16  1   function mag (a:complex): real;
    46:C        2     begin  mag := sqrt(sqr(a.re)+sqr(a.im))  end;
    47:S
    48:D   -24  1   function scmul (scale:real; a:complex): complex;
    49:C        2     begin
    50:C        2       scmul.re := scale*a.re;
    51:C        2       scmul.im := scale*a.im
    52:C        2     end;
    53:S
    54:C        1 end. (*complexmath*)

No errors.
```

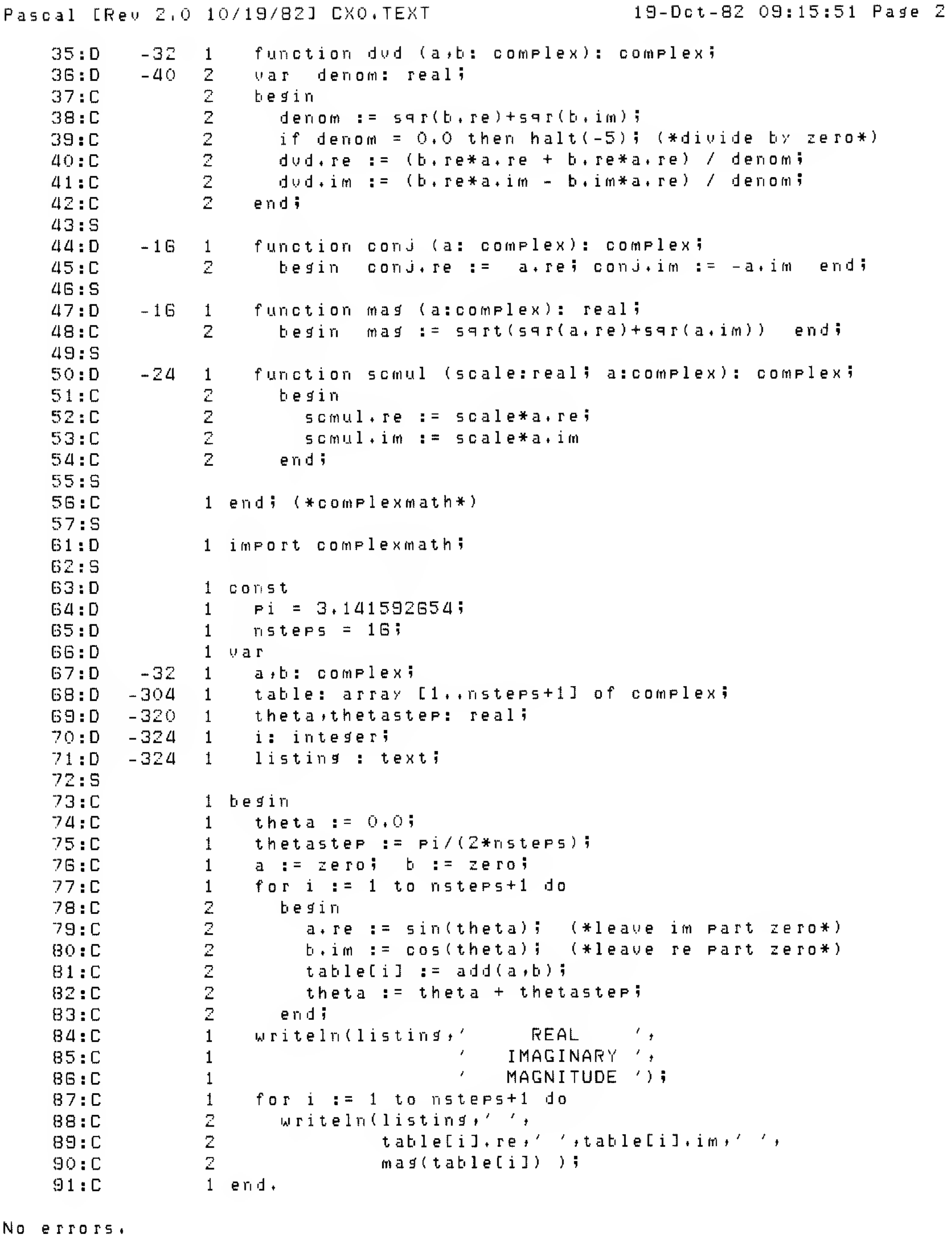
Developing and Testing a Module

The Workstation environment supports a structured approach to the development and testing of software modules. This is important because modules often become part of the system library, and many programs may depend on them. The usual steps in the development cycle are:

- Decide what the module will do – define its functionality. Write the interface part first, specifying what other modules will be needed and what things the module will export. Remember that when the finished module is imported into a program, only this interface will be "visible". Figure out how a program will use the exported things to get the module to do its job.
- Decide how the module will be tested. Write a test program which will thoroughly exercise it.
- Write the implement part of the module. Embed the completed module in the test program, and compile the two together. Leave the module inside the program until you're satisfied with the results.
- Extract the module from the test program. This can be done by using the Librarian to pull it out of the compiled test program, or by separating the module's source text with the Editor and compiling it independently.
- Use the compiled module. It can be put in the current system library (which is normally the LIBRARY file), or left as a user library which is manually linked to dependent programs, or loaded into memory by the Permanent load command.

The following listing shows the source of CXMODULE embedded into the program called CX (also on the DOC: disc):

```
Pascal [Rev 2.0 10/19/82] CXO.TEXT                 19-Oct-82 09:15:51 Page 1

    1:D          0 program cx (listing);
    2:S
    3:D          1 module complexmath;
    4:D          1 export
    5:D          1   type
    6:D          1     complex = record
    7:D          1                  re:  real;
    8:D          1                  im:  real;
    9:D          1               end;
   10:D          1   const
   11:D          1     zero = complex [re:0.0,im:0.0];
   12:S
   13:D          1   function equal (a,b: complex): boolean;
   14:D          1   function add   (a,b: complex): complex;
   15:D          1   function mul   (a,b: complex): complex;
   16:D          1   function dvd   (a,b: complex): complex;
   17:D          1   function conj  (a: complex):   complex;
   18:D          1   function mag   (a: complex):   real;
   19:D          1   function scmul (scale:real; a:complex): complex;
   20:S
   21:D          1 implement
   22:D          1
   23:D     -32  1   function equal (a,b: complex): boolean;
   24:C          2     begin equal := (a.re=b.re) and (a.im=b.im)  end;
   25:S
   26:D     -32  1   function add (a,b: complex): complex;
   27:C          2     begin add.re := a.re+b.re; add.im := a.im+b.im  end;
   28:S
   29:D     -32  1   function mul (a,b: complex): complex;
   30:C          2     begin
   31:C          2       mul.re := (a.re*b.re-a.im*b.im);
   32:C          2       mul.im := (a.re*b.im+a.im*b.re);
   33:C          2     end;
   34:S
```



```
Pascal [Rev 2.0 10/19/82] CXO.TEXT                19-Oct-82 09:15:51 Page 2

     35:D    -32  1    function dvd (a,b: complex): complex;
     36:D    -40  2    var  denom: real;
     37:C         2    begin
     38:C         2      denom := sqr(b.re)+sqr(b.im);
     39:C         2      if denom = 0.0 then halt(-5); (*divide by zero*)
     40:C         2      dvd.re := (b.re*a.re + b.re*a.re) / denom;
     41:C         2      dvd.im := (b.re*a.im - b.im*a.re) / denom;
     42:C         2    end;
     43:S
     44:D    -16  1    function conj (a: complex): complex;
     45:C         2      begin  conj.re :=  a.re; conj.im := -a.im  end;
     46:S
     47:D    -16  1    function mag (a:complex): real;
     48:C         2      begin  mag := sqrt(sqr(a.re)+sqr(a.im))  end;
     49:S
     50:D    -24  1    function scmul (scale:real; a:complex): complex;
     51:C         2      begin
     52:C         2        scmul.re := scale*a.re;
     53:C         2        scmul.im := scale*a.im
     54:C         2      end;
     55:S
     56:C         1  end; (*complexmath*)
     57:S
     61:D         1  import complexmath;
     62:S
     63:D         1  const
     64:D         1    pi = 3.141592654;
     65:D         1    nsteps = 16;
     66:D         1  var
     67:D    -32  1    a,b: complex;
     68:D   -304  1    table: array [1..nsteps+1] of complex;
     69:D   -320  1    theta,thetastep: real;
     70:D   -324  1    i: integer;
     71:D   -324  1    listing : text;
     72:S
     73:C         1  begin
     74:C         1    theta := 0.0;
     75:C         1    thetastep := pi/(2*nsteps);
     76:C         1    a := zero;  b := zero;
     77:C         1    for i := 1 to nsteps+1 do
     78:C         2      begin
     79:C         2        a.re := sin(theta);  (*leave im part zero*)
     80:C         2        b.im := cos(theta);  (*leave re part zero*)
     81:C         2        table[i] := add(a,b);
     82:C         2        theta := theta + thetastep;
     83:C         2      end;
     84:C         1    writeln(listing,'      REAL     ',
     85:C         1                    '    IMAGINARY ',
     86:C         1                    '    MAGNITUDE ');
     87:C         1    for i := 1 to nsteps+1 do
     88:C         2      writeln(listing,' ',
     89:C         2              table[i].re,' ',table[i].im,' ',
     90:C         2              mag(table[i]) );
     91:C         1  end.

No errors.
```

```
     REAL        IMAGINARY     MAGNITUDE
 0.00000E+000 1.00000E+000 1.00000E+000
 9.80171E-002 9.95185E-001 1.00000E+000
 1.95090E-001 9.80785E-001 1.00000E+000
 2.90285E-001 9.56940E-001 1.00000E+000
 3.82683E-001 9.23880E-001 1.00000E+000
 4.71397E-001 8.81921E-001 1.00000E+000
 5.55570E-001 8.31470E-001 1.00000E+000
 6.34393E-001 7.73010E-001 1.00000E+000
 7.07107E-001 7.07107E-001 1.00000E+000
 7.73010E-001 6.34393E-001 1.00000E+000
 8.31470E-001 5.55570E-001 1.00000E+000
 8.81921E-001 4.71397E-001 1.00000E+000
 9.23880E-001 3.82683E-001 1.00000E+000
 9.56940E-001 2.90285E-001 1.00000E+000
 9.80785E-001 1.95090E-001 1.00000E+000
 9.95185E-001 9.80171E-002 1.00000E+000
 1.00000E+000 -2.0510E-010 1.00000E+000
```

An Illustration

The accompanying listing shows the module "complexmath" embedded in a test program. The test program isn't very thorough, since it only checks the constant "zero" and the "add" and "mag" functions.

Modules embedded in a program may be intermixed with global constant, type, and variable declarations, but all modules must appear before any of the program's global procedures and functions. Usually all the modules are put first, followed by the program's own globals. If there are several modules, they must be ordered so that no module is imported by another (or by the program) until it has been declared.

Notice the semicolon following the END of the module (line 56), and that the program must have an IMPORT declaration (line 61) even though the module is physically present in the program.

Program "cx" can be compiled and run as shown. If you'd like to try it, invoke the Compiler by pressing (C) at the Main Command Level. When asked what text to compile, put the disc labelled DOC: in a drive and answer:

```
DOC:CX (ENTER)
```

Let the Compiler put the output file on the same disc (accept the default output file).

Compiling a Module Separately

The file generated by compiling CX.TEXT is a library with two modules, the main program "cx" and module "complexmath". Strictly speaking a program isn't a module, but within a library it has a directory entry just as if it were. You might wish to use the Librarian and see this for yourself. The Librarian can display every detail of a code file. Had there been several modules in "cx", each one would have had a separate directory entry.

It's important to be clear about the distinction between modules and libraries. A library is a file that contains object-code module(s); it is created by the Compiler or Assembler or Librarian. The library's name is its file name, which you can see with the Filer. Inside the library is a directory naming all the modules in that file. The library directory can only be displayed by the Librarian.

If you were satisfied at this point with the testing of "complexmath", you could use the Librarian to pull that one module out of the code file, and make it a user library or add it to the system library. The Librarian chapter describes how to do this.

Another alternative is to compile the module separately. Simply use the Editor to create a text file having only the module. Notice that when the module is compiled alone, it must be followed by a period instead of a semicolon. The Compiler will also accept a sequence of several modules, separated by semicolons. The last one must be followed by a period. The program listing "Complex Using $SEARCH", below, shows the listing generated by a separate compilation.

How the Compiler Finds Library Modules

A module which has been compiled is called a "library module". Library modules can be imported by programs or other modules, because the compiled code file carries with it a description of the module's interface. The Compiler is able to read this description and from it determine how to properly access everything exported by the module. (Note that modules which have been "Linked" by the Librarian do not contain the module interface descriptions.)

When the Compiler processes an IMPORT declaration, it must find the modules named in the import list and read their interface specifications. A particular search pattern is followed, which is repeated for each module named in the list.

- If the imported module has been previously declared or imported in the source text being compiled, the reference is to that module.
- If nomodule of that name has been found, the Compiler must search library files on mass storage. The files to be searched may be specified by a $SEARCH$ option. See the Compiler options section of this chapter.
- If there is no $SEARCH$ option or the module is not found in the specified list of files, the Compiler goes on to look in the current system library.
- If the module still isn't found, error 104 (undeclared identifier) is issued.

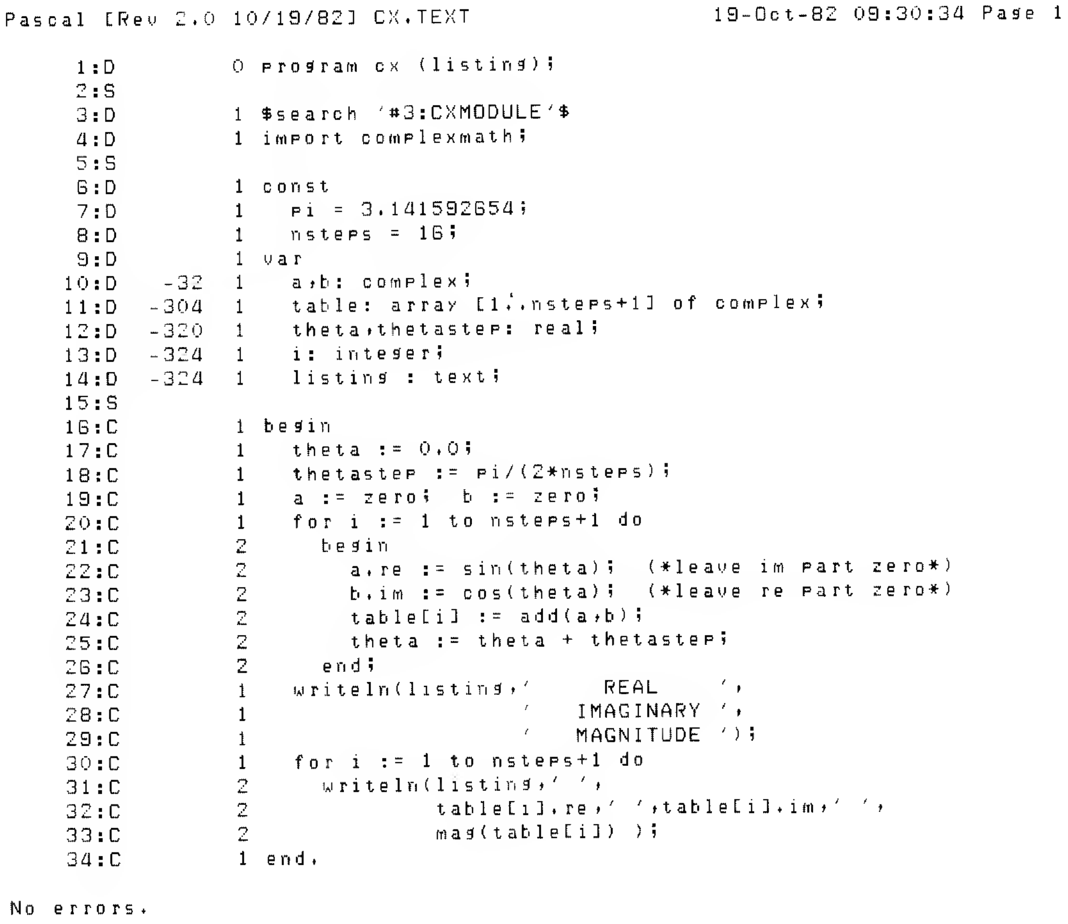
Note

The Compiler **does not** search libraries which have been loaded into memory with the P-load command. Module interface specifications are not retained with memory-resident libraries.

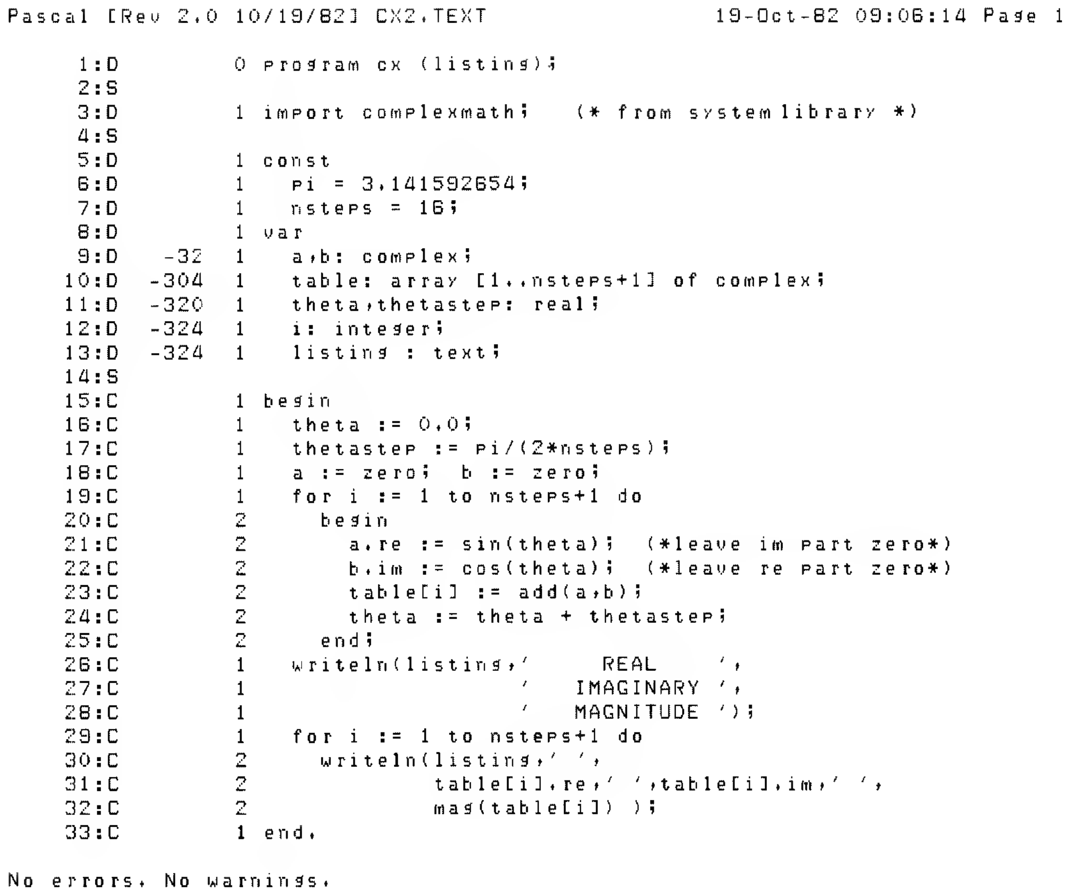
A module which is imported may itself import other modules, which are listed in its import section. The Compiler must follow such a chain all the way back to its root, to a module which imports no others. The earch pattern just described is applied recursively, to a maximum depth of ten levels. For a restriction, see the section below on $INCLUDE files. Sometimes in following an import chain, a module is named in more than one import list. The Compiler actually reads the interface specification for a module just one.

If a program imports module "A", which in turn imports module "B", the things exported from "B" are nevertheless hidden from the program. To make them visible, "B" must also be imported into the program.

The listing below shows program "cx" recompiled to search for module "complexmath" in a library called "CXMODULE" on mass storage unit #3. The second listing shows "cx" recompiled assuming "complexmath" has been put into the current System Library.



```
Pascal [Rev 2.0 10/19/82] CX.TEXT                   19-Oct-82 09:30:34 Page 1

     1:D          0 program cx (listing);
     2:S
     3:D          1 $search '#3:CXMODULE'$
     4:D          1 import complexmath;
     5:S
     6:D          1 const
     7:D          1   pi = 3.141592654;
     8:D          1   nsteps = 16;
     9:D          1 var
    10:D    -32   1   a,b: complex;
    11:D   -304   1   table: array [1..nsteps+1] of complex;
    12:D   -320   1   theta,thetastep: real;
    13:D   -324   1   i: integer;
    14:D   -324   1   listing : text;
    15:S
    16:C          1 begin
    17:C          1   theta := 0.0;
    18:C          1   thetastep := pi/(2*nsteps);
    19:C          1   a := zero;  b := zero;
    20:C          1   for i := 1 to nsteps+1 do
    21:C          2     begin
    22:C          2       a.re := sin(theta);  (*leave im part zero*)
    23:C          2       b.im := cos(theta);  (*leave re part zero*)
    24:C          2       table[i] := add(a,b);
    25:C          2       theta := theta + thetastep;
    26:C          2     end;
    27:C          1   writeln(listing,'     REAL      ',
    28:C          1                   '   IMAGINARY ',
    29:C          1                   '   MAGNITUDE ');
    30:C          1   for i := 1 to nsteps+1 do
    31:C          2     writeln(listing,' ',
    32:C          2             table[i].re,' ',table[i].im,' ',
    33:C          2             mag(table[i]) );
    34:C          1 end.

No errors.
```



```
Pascal [Rev 2.0 10/19/82] CX2.TEXT                19-Oct-82 09:06:14 Page 1

    1:D          0 program cx (listing);
    2:S
    3:D          1 import complexmath;    (* from system library *)
    4:S
    5:D          1 const
    6:D          1   pi = 3.141592654;
    7:D          1   nsteps = 16;
    8:D          1 var
    9:D     -32  1   a,b: complex;
   10:D    -304  1   table: array [1..nsteps+1] of complex;
   11:D    -320  1   theta,thetastep: real;
   12:D    -324  1   i: integer;
   13:D    -324  1   listing : text;
   14:S
   15:C          1 begin
   16:C          1   theta := 0.0;
   17:C          1   thetastep := pi/(2*nsteps);
   18:C          1   a := zero;  b := zero;
   19:C          1   for i := 1 to nsteps+1 do
   20:C          2     begin
   21:C          2       a.re := sin(theta);  (*leave im part zero*)
   22:C          2       b.im := cos(theta);  (*leave re part zero*)
   23:C          2       table[i] := add(a,b);
   24:C          2       theta := theta + thetastep;
   25:C          2     end;
   26:C          1   writeln(listing,'      REAL     ',
   27:C          1                   '   IMAGINARY  ',
   28:C          1                   '   MAGNITUDE  ');
   29:C          1   for i := 1 to nsteps+1 do
   30:C          2     writeln(listing,' ',
   31:C          2             table[i].re,' ',table[i].im,' ',
   32:C          2             mag(table[i]) );
   33:C          1 end.

No errors. No warnings.
```

How the Loader Finds Library Modules

When the Compiler processes an import declaration, it does not copy, or in any other way bind the library module into the program being compiled. Instead it emits reference information (called REF's) which enables the loader or linker to make the required connections later. Usually REF's are satisfied (hooked up to the library module) at the last possible moment: when you Run the program.

A compiled program contains no record of where the Compiler found any imported modules. The loader has a search pattern it uses to find imported things the program needs:

- First, the file being loaded is searched. There may be modules in it which were compiled at the same time as the program.
- Then memory-resident libraries are searched. The memory-resident libraries are those you have loaded with the P command, the contents of INITLIB (which is automatically loaded at boot time), and the modules of the Operating System itself. The order of search is most-recently-loaded first.
- Finally, the current System Libraryis searched. If a required module is in the System Library, it will be loaded with the program and will remain in memory until a different program is executed.
- If there are still unresolved references, the loader reports them on the CRT. The program won't run. Control is returned to the Main Command Level.

If your program only imports from the System Library, everything is taken care of automatically. This is the most common case. If the program imports from user libraries via the $SEARCH$ option, you must help out the loader in one of three ways.:

- Use the P-load command to load copies of the libraries into memory before running the program. Do this just once, because the loader does not check to see if modules have already been loaded! Memory-resident libraries stay there until you re-boot.
- Use the Librarian to make a new library containing the compiled program and any modules it needs. This new library is an unlinked, executable program. It will automatically be linked when it is loaded.
- Use the Librarian to link the necessary modules to the program. The resulting library is a linked, executable program. It will probably still have some unresolved references (for instance to the system read and write routines), which will be resolved at load time.

A Subtle Point

The loader doesn't search for modules, it searches for **external names**. Each procedure or function exported has an external name, as do most structured constants. A single name is used for all the variables a module exports; it is actually the name of a place in memory where storage for the variables will be allocated. Certain things, such as types and simple constants, are only useful at compile time and so have no external name.

If two modules which are loaded define the same load-time name, the most recently loaded copy overrides the older one. Generally this makes no difference, because external names created by the Compiler identify the module where the name originated. However, some module names are used by the Operating System. You should avoid using these names from your own modules unless you **intend** to override the name of a system entry point. These names are listed in the Technical Reference Appendix.

$INCLUDE Files

The source text of a module or program can be broken up into several text files which are edited separately but compiled as a group. The $INCLUDE option tells the Compiler to insert the Pascal source of another file into the one it is presently compiling.

```
program showinclude (input,output);
$include 'MYVOL:DECLARS'$
$include 'SYSVOL:BODY'$
end.
```

If the required volume is not on-line when needed, the Compiler pauses and prompts you to insert the proper volume.

Miscellaneous

- An included file may in turn include another file. This "nesting" is allowed to a maximum depth of 10.
- Importing a library module is a form of file inclusion, and counts against the maximum allowable depth of 10 while the import declaration is being processed.
- If the imported module has an import declaration in its own interface, the Compiler will follow the chain and find those module interfaces too. This is another form of nested file inclusion.

What Can Go Wrong?

This section discusses some problems which may occur when using the Compiler, and how to solve them.

Can't Run the Compiler

1. If the system reports, `Cannot open 'CMP:COMPILER'`, the volume with the Compiler is not online. You may have removed the volume and not put it back or changed it with the What command. If the Compiler wasn't found when the system booted, you are expected to put the CMP: disc (which contains the Compiler) on-line.
2. If the system reports, `Cannot load 'COMPILER'`, either the disc medium is bad, or not enough memory is installed in the Computer to run the Compiler. It is desirable to have at least 393K bytes; the system is normally sold with at least 524K bytes.

Errors 900 thru 908

During compilation, three files are written by the Compiler: the code file, which is the one you want, and the REF and DEF files. The latter two are temporary working storage for linkage information which is appended to the code file if the compilation terminates normally. All three of these files are normally opened on the same volume (the volume to which you directed the code file).

Each of these files is subject to three classes of error:

- Error in opening the file.
- Insufficient space to open the file.
- File fills up before compilation finishes.

An error in opening the file usually means the volume is not online. It can also indicate that the volume's directory is full.

The amount of space allocated to the code file is usually half of the largest free area on the volume, with the potential to expand to the second half of that area if needed. If you get errors 900, 903, or 906 you need to make more room on the volume to which the code file was directed, or use a different volume.

The REF file by default is opened with 30 blocks of disk space on the same volume as the code file. A Compiler option at the beginning of the source program can change the size and the volume selected for REF. There's no simple rule which gives the "right" size for the REF file. If the file fills up (error 907), make it bigger in proportion to the amount of program that remained to compile when the error occurred.

`$REF 50$`	Allocate 50 blocks
`$REF 'CHARLIE:'$`	Put it on volume CHARLIE
`$REF 'CHARLIE:', REF 50$`	Put it on CHARLIE and allocate 50 blocks

Exactly analogous remarks hold for the DEF file, except that its default size is 10 blocks and the Compiler option is DEF .

Errors When Importing Library Modules

1. Syntax errors in the interface of an imported library module. This usually indicates that the library module itself tried to import some other module which was not found by the Compiler's search algorithm.
2. Errors 608, 610: Include or import nesting too deep. If module "A" imports "B", which imports "C" and so forth, the Compiler must follow the chain to its end. The chain can only be 10 imports deep. Since the same file handling mechanism is also used to process $INCLUDEd files, the combined limit on import and inclusion nesting is 10 deep.
3. Error 613: Imported module does not have interface text. If the library has been linked by the Librarian, the interface specification has been removed. Also, a main program looks internally like a module; but it has no interface text.

Not Enough Memory

If the Compiler generates error -2 "Not Enough Memory", there isn't enough room in memory to compile the program. You can watch the numbers which appear on the screen in square brackets as the compilation proceeds -- they show approximately how much memory is left. There are two primary reasons for running out of memory during a compilation. One of them is large procedure bodies, and the other is P-loaded files.

Large Procedure Bodies

When the Compiler processes a procedure, the entire procedure (declarations and body) is scanned. An internal representation of the procedure, called a "tree", is built. This tree is not complete until the scanner reaches the end of the procedure, and only then does code generation begin. The tree form takes a lot of storage, particularly the statements making up the body. If you write a procedure whose body is ten pages long, the Compiler is very likely to run out of memory. The moral is that you should keep your procedures reasonably short. A good guideline is that no procedure should be longer than a page or two.

P-loaded Files

If you've Permanent-loaded a lot of libraries or programs, or space has been allocated to a memory-resident mass storage volume, you can reboot the system to recover the memory, and try again.

Insufficient Space for Global Variables

You may discover, either at compile time or at run time, that there isn't sufficient space for the global variables of your program. If this happens, please refer to "Implementation Restrictions" in this chapter, which explains the limitations and what to do if you exceed them.

Errors 403 thru 409

These errors should never be reported. They indicate a malfunction in the Compiler itself. If this ever happens, please show the program which causes it to your HP field support contact.

Compiler Options

Compiler options affect the code emitted by the Compiler. For instance, the $DEBUG ON$ option causes the Compiler to emit a TRAP instruction after the object code for each Pascal statement, allowing you to single-step the program.

Sometimes there are restrictions on where an option may appear:

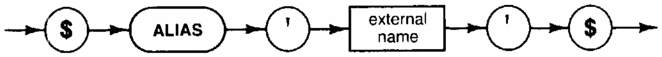
Location	Restrictions
Anywhere	Indicates that the location of the option in the file is irrelevant.
At front	Applies to entire source file; must appear before the first "token" in the source file (before PROGRAM, or before MODULE if compiling a list of modules).
Not in body	Applies to a whole procedure or function; can't appear between BEGIN and END. Good practice to put these options immediately before the word BEGIN or the procedure heading.
Statement	Can be applied on a statement-by-statement basis or to a group of statements, by enabling before and disabling after the statement(s) of interest.
Special	As explained under the particular option.

If an option appears in the interface (import or export) part of a module, it will have effect as the module is compiled. However, the option itself will not become part of the interface specification in the compiled module's object code.

ALIAS

Default: External name = Procedure Name
Location: Special, See Below

This Compiler option causes a name, other than the name used in the Pascal procedure or function declaration, to be used by the loader.



Item	Description/Default	Range Restrictions
external name	string	1 to 80 ASCII Characters

Semantics

The string parameter specifies the external name for the procedure in whose header the option appears.

Example:

```
    procedure $alias 'charlie'$ p (i: integer);  external;
```

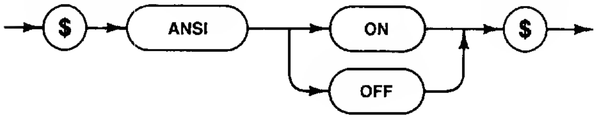
Within the program, calls use the name "p"; but the loader will link to a physical routine called "charlie".

Should appear between the keywords PROCEDURE or FUNCTION and the routine's identifier.

ANSI

Default: OFF
Location: At Front

This Compiler option selects whether an error message is to be emitted for use of any feature of HP Standard Pascal not contained in ANSI/ISO Standard Pascal.



Semantics

"ANSI" is interpreted as "ANSI ON".

ON causes error messages to be issued for use of any feature of HP Standard Pascal which is not part of ANSI/ISO Standard Pascal.

OFF suppresses the error messages.

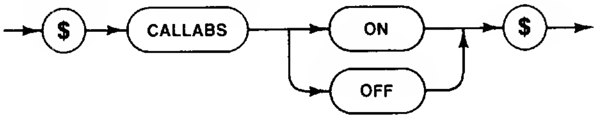
Example:

```
    $ansi on$
```

CALLABS

Default: ON
Location: Statement

This Compiler option determines whether 16-bit relative or 32-bit absolute jumps are to be generated by the Compiler.



Semantics

"CALLABS" is interpreted as "CALLABS ON".

ON specifies that 32-bit absolute jumps will be emitted for all forward and external procedure calls.

OFF specifies 16-bit PC-relative jumps.

Allowed on a statement-by-statement basis.

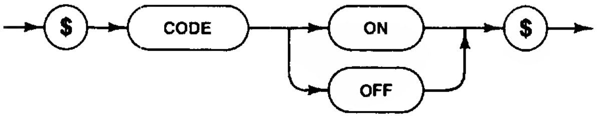
Example:

```
$callabs off$
```

CODE

Default: ON
Location: Not in Body

This Compiler option is used to control whether a CODE file will be generated by the Compiler.



Semantics

"CODE" is interpreted as "CODE ON".

ON specifies that a code file will be generated.

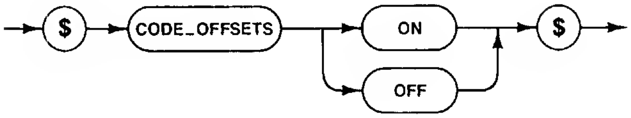
Example:

```
    $code off$
```

CODE_OFFSETS

Default: OFF
Location: Not in Body

This Compiler option controls the inclusion of program counter offsets in the compiler listing.



Semantics

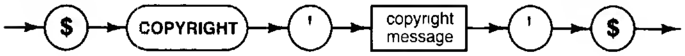
"CODE_OFFSETS" is interpreted as "CODE_OFFSETS ON".

ON specifies that line-number/program-counter pairs will be printed for each executable statement listed. This can be applied on a procedure-by-procedure basis.

COPYRIGHT

Default: Not Applicable
Location: Anywhere

This Compiler option is provided for inclusion of copyright information.



Item	Description/Default	Range Restrictions
copyright message	string	1 to 80 ASCII Characters

Semantics

The string parameter is placed in the object file as the owner of the copyright. If more than one COPYRIGHT option is included, the last one is effective.

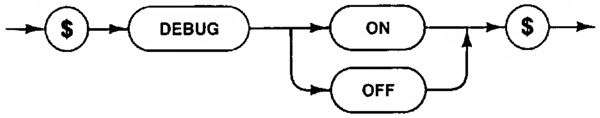
Example:

```
    $copyright 'Hewlett Packard Company, 1981'$
```

DEBUG

Default: OFF
Location: Not in Body

This Compiler option controls whether the code produced by the Compiler contains the additional information necessary for full use of the Debugger system.



Semantics

"DEBUG" is interpreted as "DEBUG ON"

"DEBUG ON" will cause debugging instructions to be emitted for the procedure bodies following it. May be applied on a procedure-by-procedure basis.

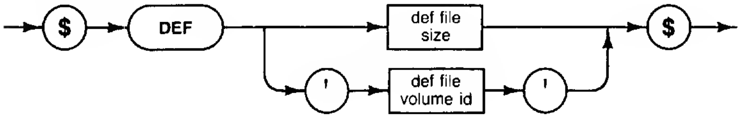
Example:

```
procedure buggy;
var i: integer;
$debug on$
begin
...
end;
$debug off$
```

DEF

Default: 10 records on same volume as code output
Location: At Front

This Compiler option allows the user to change the size and location of the temporary Compiler file named ".DEF".



Item	Description/Default	Range Restrictions
def file size	integer constant	less than 32767
def file volume id	string	valid volume id (see glossary)

Semantics

If the parameter is a string, it specifies the volume where a temporary Compiler file called ".DEF", which holds external definitions, will be stored. If the parameter is a number, it specifies how many logical records will be allocated for the DEF file. See "What Can Go Wrong, Errors 900 to 908".

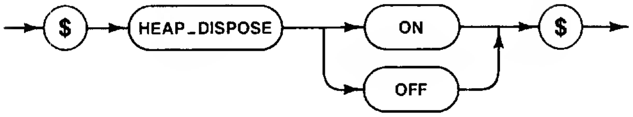
Examples:

```
$def 50$
$def 'JunKvol:'$
$def 'JunKvol:', def 50$
```

HEAP_DISPOSE

Default: OFF
Location: At Front

This Compiler option enables and disables "garbage collection" in the heap.



Semantics

"HEAP_DISPOSE" is interpreted as "HEAP_DISPOSE ON"

ON indicates that DISPOSE allows disposed objects to be reused.

OFF does not recycle disposed objects. If enabled, this option must appear at the front of the **main program**.

Example:

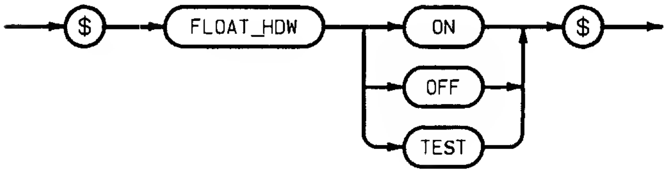
```
$heap_dispose on$
program recycle;
...
begin
  dispose(p);  (*free up cell*)
  new(p);      (*probably gets same cell back*)
end.
```

The HEAP_DISPOSE option must be the same (either ON or OFF) in the program and **all** modules imported by the program. Erroneous results may occur if those declarations don't agree, because there is no way for the Compiler to check on which option other modules have used.

FLOAT_HDW

Default: OFF
Location: Not in body

This option enables and disables the use of floating-point hardware.



Semantics

An optional floating-point hardware board (HP 98635) is available for Series 200 Computers to increase the execution speed of floating-point math programs.

"FLOAT_HDW" is interpreted as "FLOAT_HDW ON"

ON instructs the Compiler to generate accesses to hardware for most floating-point operations. If the hardware does not exist when the program is executed, an error will result.

OFF tells the Compiler to generate calls to libraries for all floating-point operations.

TEST causes the Compiler to generate both hardware accesses and library calls. The Compiler automatically includes code to test for the presence of floating-point hardware. At execution time, if the test succeeds, the hardware accesses are used, otherwise the library calls are used.

The operations that use the hardware include: addition, subtraction, multiplication, division, negation, and `sqr` function. All other math functions call library routines. There are libraries that access the floating-point hardware. Hardware can also be used by any operation that converts an integer to a real or longreal. The hardware is not used by operations that convert reals or longreals into integers.

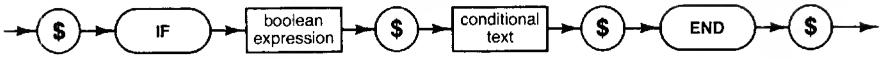
Example

```
    $float_hdw test$
```

IF

Default: Not Applicable
Location: Anywhere

This Compiler option allows conditional compilation.



Item	Description/Default	Restrictions
boolean expression	expression that evaluates to either TRUE or FALSE	may only contain compile time constants
conditional text	source to be conditionally compiled	

If the expression evaluates to FALSE, then text following the option is skipped up to the next END option.

If the boolean expression evaluates to TRUE, then the text following the option is compiled normally.

IF-END option blocks may not be nested.

String constants may not be used.

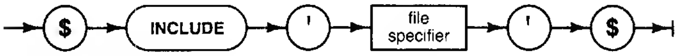
Example:

```
const fancy = true;
      limit = 10;
      size = 9;
...
$if fancy and ((size+1)<limit)$
  ...   (* this will be skipped *)
$end$
...
$if FALSE$
...(* this will also be skipped*)
$end$
```

INCLUDE

Default: Not Applicable
Location: Anywhere

This Compiler option allows text from another file to be included in the compilation process.



Item	Description/Default	Range Restrictions
file specifier	string	any valid file specifier (see Glossary)

The string parameter names a file which contains Pascal source to be included at the current position in the program. Included code may contain additional INCLUDE options (nesting level is 10). The remainder of the line which contains this option must be blank except for the closing $.

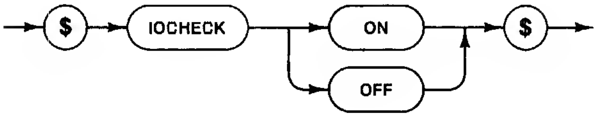
Example:

```
program inclusive;
$include 'source:declars'$
$include 'source:body'$
end.
```

IOCHECK

Default: ON
Location: Statement

This Compiler option enables and disables error checking following calls to system I/O routines.



Semantics

"IOCHECK" is interpreted as "IOCHECK ON"

ON specifies that error checks will be emitted following calls on system I/O routines such as RESET, REWRITE, READ, WRITE. Can be used in conjunction with the standard function IORESULT if UCSD or SYSPROG language features have been enabled. Allowed on a statement-by-statement basis.

OFF specifies that no error will be reported in case of failure.

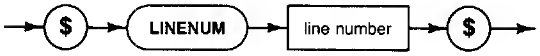
Example:

```
$ucsd$
...
$iocheck off$
reset(f,'datafile');
$iocheck on$
if ioresult <> 0 then writeln('IO error');
```

LINENUM

Default: Not Applicable
Location: Anywhere

This Compiler option allows the user to establish an arbitrary line number value.



Item	Description/Default	Range Restrictions
line number	integer numeric constant	0..65535

Semantics

The integer parameter becomes the current line number (for listing and debugging purposes).

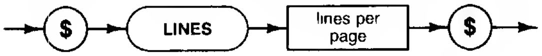
Example:

```
$linenum 20000$
```

LINES

Default: 60 lines per page
Location: Anywhere

This Compiler option allows the user to specify the number of lines-per-page on the Compiler listing. 2000000 lines-per-page suppresses autopagination.



Item	Description/Default	Range Restrictions
lines per page	integer numeric constant	20 thru MAXINT

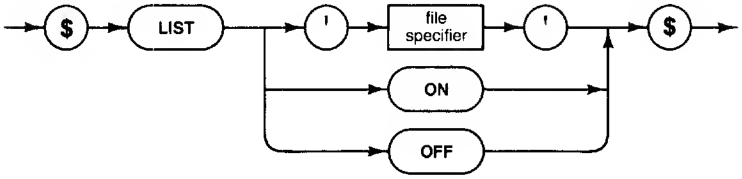
Example:

```
$lines 55$
$lines 2000000$ (*suppress autopagination*)
```

LIST

Default: ON to PRINTER:
Location: Anywhere

This Compiler option controls whether or not a listing is being generated, and to where it will be directed.



Item	Description/Default	Range Restrictions
file specifier	string	any valid file specifier (see glossary)

Semantics

"LIST" is interpreted as "LIST ON".

LIST with a file specifier specifies that the file is to receive the compilation listing.

LIST OFF suppresses listing.

LIST ON resumes listing. No listing will be produced at all, regardless of this option, unless requested by the operator when the Compiler is invoked.

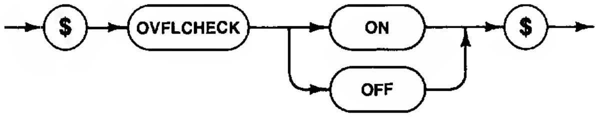
Examples:

```
$list 'myvol:keeplist.text'$
$list 'printer:'$
$list off$
```

OVFLCHECK

Default: ON
Location: Statement-by-statement

This Compiler option gives the user some control over overflow checks on arithmetic operations.



Semantics

"OVFLCHECK" is interpreted as "OVFLCHECK ON"

ON specifies that overflow checks will be emitted for all in-line arithmetic operations.

OFF does not suppress all checks; they will still be made for 32-bit integer DIV, MOD, and multiplication.

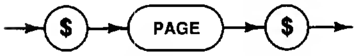
Example:

```
    $ovflcheck off$
```

PAGE

Default: Not Applicable
Location: Anywhere

This Compiler option causes a formfeed to be sent to the listing file if compilation listing is enabled.



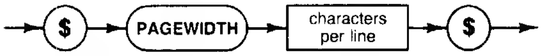
Example:

```
$page$
```

PAGEWIDTH

Default: 120
Location: Anywhere

This Compiler option allows the user to specify the width of the compilation listing.



Item	Description/Default	Range Restrictions
characters per line	integer numeric constant	80 thru 132

Semantics

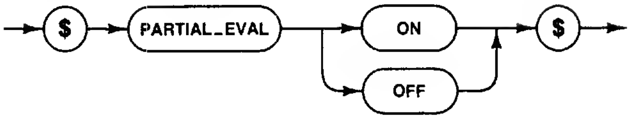
The integer parameter specifies the number of characters in a printer line.

Example:

```
$pagewidth 80$
```

PARTIAL_EVAL

Default: OFF
Location: Statement-by-statement



Semantics

"PARTIAL_EVAL" is interpreted as "PARTIAL_EVAL ON".

ON suppresses the evaluation of the right operand of the AND operator when the left operand is FALSE. The right operand will not be evaluated for OR if the left operand is TRUE.

OFF causes all operands in logical operations to be evaluated regardless of the condition of any other operands.

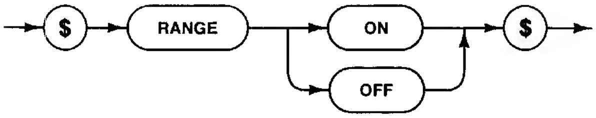
Example:

```
$partial_eval on$
while (p<>nil) and (p^.count>0) do
  p := p^.link;
```

RANGE

Default: ON
Location: Statement-by-statement

This Compiler option enables and disables run-time-checks for range errors.



Semantics

"RANGE" is interpreted as "RANGE ON".

ON specifies that run time checks will be emitted for array and case indexing, subrange assignment, and pointer dereferencing.

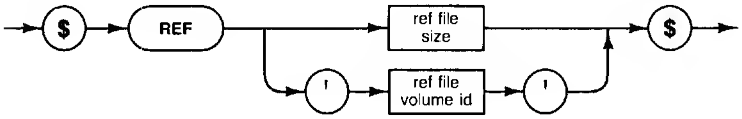
Example:

```
    var a: array[1..10] of integer;
        i: integer;
    ...
    i := 11;
    $range off$
    a[i] := 0;    (* invalid index not caught! *)
```

REF

Default: 30 records on same volume as code output
Location: At Front

This Compiler option allows the user to change the size and location of the temporary Compiler file named ".REF".



Item	Description/Default	Range Restrictions
ref file size	integer numeric constant	less than 32767
ref file volume id	string	valid volume id (see glossary)

Semantics

If the parameter is a string, it specifies the volume where a temporary Compiler file called ".REF", which holds external references, will be stored. If the parameter is a number, it specifies how many logical records will be allocated for the REF file. See "What Can Go Wrong, Errors 900 to 908".

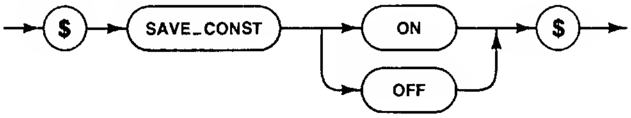
Examples:

```
$ref 20$
$ref 'JUNKVOL:'$
$ref 'JUNKVOL:', ref 50$
```

SAVE_CONST

Default: ON
Location: Anywhere

This Compiler option controls whether the name of a structured constant may be used by other structured constants.



Semantics

"SAVE_CONST" is interpreted as "SAVE_CONST ON".

ON specifies that compile-time storage for the value of each structured constant will be retained for the scope of the constant's name (so that other structured constants may use the name).

OFF specifies that storage will be deallocated after code is generated for the structured constant.

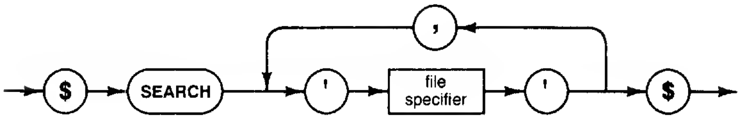
Example:

```
$save_const off$
type ary = array [1..100] of integer;
const acon = ary [345,45691, ..... ];
    (*big constants take lots of compile-time
      memory*)
```

SEARCH

Default: Not Applicable
Location: Anywhere

This Compiler option is used to specify files to be used to satisfy IMPORT declarations.



Item	Description/Default	Range Restrictions
file specifier	string	any valid file specifier (see Glossary)

Semantics

Each string specifies a file which may be used to satisfy IMPORT declarations. Files will be searched in the order given. The current system library is always searched last. A maximum of 9 files may be listed.

Multiple SEARCH options are allowed; for instance, you may want to use one for each import declaration. Note that only the last one encountered during compilation will be in effect for any import statement (i.e., these options are not cumulative).

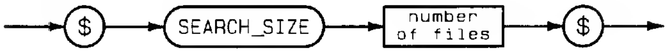
Example:

```
$search 'FIRSTFILE','SECONDFILE'$
```

SEARCH_SIZE

Default:	10 files
Location:	At front

This Compiler option allows you to increase the number of external files you may SEARCH during a module's compilation.



Item	Description/Default	Range Restrictions
number of files	integer numeric constant	less than 32767

Semantics

When compiling a Pascal module, it is sometimes desirable to import another module from another file. To import a module from another file, the SEARCH option is used to identify the file. Up to ten SEARCH options may be given unless the SEARCH_SIZE options is given. The SEARCH_SIZE option allows you to SEARCH up to 32,766 external files for imported modules.

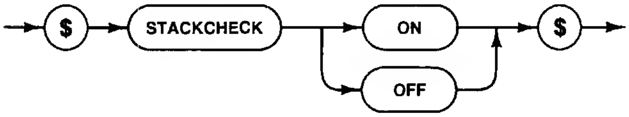
Example:

```
    $search_size 30$
```

STACKCHECK

Default: ON
Location: Not in Body

This Compiler option enables and disables stack overflow checks.



Semantics

"STACKCHECK" is interpreted as "STACKCHECK ON".

ON specifies that stack overflow checks will be generated at procedure entry. It is very dangerous to turn overflow checks off! Obscure and unreported errors may result.

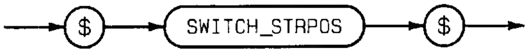
Example:

```
$stackcheck off$
procedure unsafe;
var
  may_smash_heap: array [1..500] of integer;
begin ... end;
```

SWITCH_STRPOS

Default:	Off
Location:	At front

This Compiler option reverses the positions of the parameters of the STRPOS function.



Semantics

When this Compiler option is used, the exprected order of the parameters is that of the HP standard. In Series 200 Pascal (like UCSD's POS function), the STRPOS function expects the first string parameter to be the search pattern and the second string parameter to be the source string in which the search takes place. Later the HP standard was established with the order of the parameters reversed. If $WARN OFF$ is not in effect, then the Compiler issues a harmless warning that you are not conforming to the standard. If you wish to conform to the standard, give the SWITCH_STRPOS option in your program.

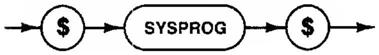
Example:

```
    $switch_strpos$
```

SYSPROG

Default: System Programming Extensions not enabled
Location: At Front

This Compiler option makes available some language extensions which are useful in systems programming applications. See ‘‘System Programming Language Features’’ in this chapter.



Semantics

$SYSPROG$ is interpreted as $SYSPROG ON$

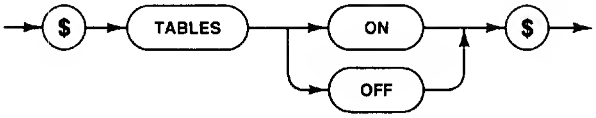
Example:

```
$sysprog$
program machinedependent;
...
```

TABLES

Default: OFF
Location: Not in Body

This Compiler option allows the user to turn on and off the listing of symbol tables.



Semantics

"TABLES" is interpreted as "TABLES ON"

ON specifies that symbol table information will be printed following the listing of each procedure. This is useful for very low-level debugging.

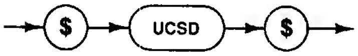
Example:

```
$tables$
procedure hasabug (var p: integer);
var
...
```

UCSD

Default: UCSD not enabled
Location: At Front

This Compiler option allows the compiler to accept most UCSD Pascal language extensions. See the subsequent "Converting UCSD Pascal" section later in this chapter.



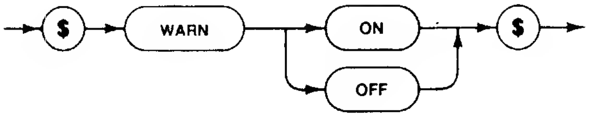
Example:

```
$ucsd$
program funnyio;
var
  f: file;    (* no type specified! *)
begin
  unitread(8,ary,80,10);
end.
```

WARN

Default: ON
Location: At Front

This option allows the user to suppress the generation of compiler warning messages.



Semantics

"Warn" is interpreted as "WARN ON" and compiler warnings will be issued.

Example

```
$warn off$
```

System Programming Language Extensions

Eight extensions to HP Pascal have been provided to support machine-dependent programming and give users better control over (or access to) the hardware. These extensions may be used in any compilation which includes the $SYSPROG ON$ option at the beginning of the text.

The extensions may not be supported by other HP Pascal implementations. The Compiler displays a warning message at the end of compilation when they are enabled (unless $WARN OFF$ is used).

Absolute Addressing of Variables

A variable may be declared as located at an absolute or symbolically named address:

```
var   ioport [416000]: char;
      assemblysymbol ['asm_external_name']: integer;
```

Each variable named in a declaration may be followed by a bracketed address specifier. An integer constant specifier gives the absolute address of the variable; this is useful for addressing IO interface hardware. A quoted string literal gives the name of a load-time symbol which will be taken as the location of the variable; such a symbol must be defined (DEF'ed) by an assembly-language module which will be loaded with the program.

Error Trapping and Simulation

The TRY-RECOVER statement and the standard function ESCAPECODE have been added to allow programmatic trapping of errors. The standard procedure ESCAPE has been added to allow the generation of soft (simulated) errors.

```
try
  <statement> ;
  <statement> ;
  ...
<statement>
recover
  <statement>
```

When TRY is executed, certain information about the state of the program is recorded in a marker called the recover-block, which is pushed on the program's stack. The recover-block includes the location of the corresponding RECOVER statement, the height of the program stack, and the location of the previous recover-block if one is active. The address of the recover-block is saved, then the statements following TRY are executed in sequence. If none of them causes an error, the RECOVER is reached, its statement is skipped, and the recover-block is popped off the stack.

But if an error occurs, the stack is restored to the state indicated by the most recent recover-block. Files are closed, and other cleanup takes place during this process. If the TRY was itself nested within another one, or within procedures called while a TRY was active, that previous recover-block becomes the active one. Then the statement following RECOVER is executed. Thus the nesting of TRYs is **dynamic**, according to calling sequence, not statically structured like nonlocal goto's which can only reach labels declared in containing scopes.

The recovery process does not "undo" the computational effects of statements executed between TRY and the error. The error simply aborts the computation, and the program continues with the RECOVER statement.

When an error has been caught, the function ESCAPECODE can be called to get the number of the error. ESCAPECODE has no parameters. It returns an integer error number selected from the error code table. System error numbers are always negative.

The programmer can simulate errors by calling the standard procedure ESCAPE(n), which sets the error code to n and starts the error sequence. By convention, programmed errors have numbers greater than zero. If an ESCAPE is not caught by a recover-block within the program, it will be reported as an error by the Operating System. Negative values are reported as standard system error messages, and positive values are reported as a halt code value. Note that HALT(n) is exactly the same as ESCAPE(n).

TRY-RECOVER statements are usually structured in the following "canonical" fashion:

```
try
....
recover
   if escapecode = (whatever you want to catch)
     then
       begin
          {recovery sequence}
       end
     else
       escape(escapecode);
```

This has the effect of ensuring that errors you **don't** want to handle get passed on out to the next recover-block, and eventually to the system.· All programs which are executed are first surrounded by the Command interpreter with a try-recover sequence. The recovery action for the system is to display an error message.

Determining the Size of Variables and Types

The size (in bytes) of a type or variable can be determined by the SIZEOF function. This also is enabled by the $UCSD$ directive.

```
    n := sizeof(variable);
    n := sizeof(typename);
```

If the variable or type is a record with variants, an optional list of tagfield constants may follow the parameter. This works like the standard Pascal procedure NEW:

```
    n := sizeof(varrec,true,blue);
```

SIZEOF is not really a function, although it looks like one; it is actually a form of compile-time constant.

Relaxed Typechecking of VAR Parameters

The ANYVAR parameter specifier in a function or procedure heading relaxes type compatibility checking when the routine is called. This is sometimes useful to allow libraries to act on a general class of objects. For instance an I/O routine may be able to enter or output an array of arbitrary size.

```
type
  buffer = array [0..maxint] of char;
var
  a1: array [2..50] of char;
  a2: array [0..99] of char;

procedure output_hpib(anyvar ary:buffer; lobound,hibound:integer);
  ....

output_hpib(a1,2,50);
output_hpib(a2,0,99);
```

ANYVAR parameters are passed by reference, not by value; that is, the address of the variable is passed. Within the procedure, the variable is treated as being of the type specified in the heading.

This can be very dangerous!

For instance, if an array of 10 elements is passed as an ANYVAR paramter which was declared to be an array of 100 elements, an error will very likely occur. The called routine has **no way** to know what you actually passed, except perhaps by means of other parameters as in the example above. ANYVAR should only be used when it's absolutely required, since it defeats the Compiler's normal type safety rules.

Programs calling routines with ANYVAR parameters should be very thoroughly debugged. Careless use of this feature can crash your system.

The ANYPTR Type

Another way to defeat type checking is with the non-standard type ANYPTR. This is a pointer type which is assigment-compatible with all other pointers, just like the constant NIL. However, variables of type ANYPTR are not bound to a base type, so they can't be dereferenced. They may only be assigned or compared to other pointers. Passing as a value parameter is a form of assignment.

```
type
   p1 = ^integer;
   p2 = ^record
           f1,f2: real;
         end;
var
   v1,v1a: p1;  v2: p2;
   anyv: anyptr;
   which: (type1,type2);
begin
  new(v1);  new(v2);
  ...
  if ... then
    begin  anyv := v1;  which := type1  end
  else
    begin  anyv := v2;  which := type2  end;
  ...
  if which = type1 then
    begin
      v1a := anyv;
      v1a^ := v1a^ + 1;
    end;
end;
```

This can be very dangerous!

The Compiler has no way to know if ANYPTR tricks were used to put a value into a normal pointer. If a pointer type which is bound to a small object has its value tricked into a pointer bound to a large object, subsequent assignment statements which dereference the tricked pointer may destroy the contents of adjacent memory locations.

Careless use of ANYPTR can crash your system. Programs using this feature must be very thoroughly debugged.

Determining the Absolute Address of a Variable

```
p := addr(variable);
p := addr(variable,offset);
```

The ADDR function returns the address of a variable in memory as a value of type ANYPTR. It accepts, as an optional second parameter, an integer "offset" expression which will be added to the address; this has the effect of pointing "offset" bytes away from where the variable begins in memory. ADDR is primarily used for building or scanning data structures whose shapes are defined at run-time rather than by normal Pascal declarations.

The ADDR function is very dangerous!

It has the same dangers described above for ANYPTRs, in addition to some of its own. Use of the "offset" can produce a pointer to almost anywhere, with concommitant dangers to the integrity of system memory.

Never use ADDR to create pointers to the local variables of a procedure or function. Storage for local variables is recovered when the routine exits, so the value returned by ADDR is ephemeral.

Careless use of the pointers returned by ADDR can crash your system. Programs using this feature must be very carefully debugged.

Procedure Variables and the Standard Procedure CALL

Sometimes it is desirable to store in a variable the name of a procedure, and then later to call that procedure. For instance, the system "Unittable" is an array which contains the name of the driver to be called to perform IO on each logical volume.

A variable of this sort is called a "procedure variable". The "type" of a procedure variable is a description of the parameter list it requires. That is, a procedure variable is bound to a particular procedure heading.

```
type  procvar = procedure (op:integer);
var   p: procvar;

procedure q(op:integer);    {identically structured parameter list}
...

p := q;          {p gets the name of q; in effect p points to q}
call(p,i);       {name of proc variable, then appropriate parameter list}
```

A procedure variable is "called" by the standard procedure CALL, which takes the procedure variable as its first parameter, and a further list of parameters just as they would be passed to a real procedure having the corresponding specification.

It is not possible to create a "function variable", that is, a variable which can hold the name of a function.

Don't assign the name of an inner (non-global) procedure to a procedure variable which isn't declared in the same block as the procedure being assigned. Such a variable might be called later, after exiting the scope in which the procedure was declared. The appropriate static link would be missing, yielding unpredictable results. See "How Pascal Programs Use the Stack", at the end of this chapter, for an explanation of static links.

The IORESULT Function

Normally the Compiler emits instructions after each IO statement to verify that the transaction completed properly. If it fails, the program is terminated with an error report.

It is possible to trap IO errors programmatically, using the TRY-RECOVER statement. The system function IORESULT can then be called to discover what went wrong with the transaction.

IOchecks and IOresults

Normally the Compiler emits instructions after each IO transaction to verify that the transaction completed properly. If it didn't, the program is terminated with an error report. The error code for all IO errors is -10.

You may wish to intercept IO errors programmatically rather than have them terminate the program. This can be done two different ways. The program or module must be compiled with the $SYSPROG$ or $UCSD$ Compiler directive at the front of the source text. These directives both make available a system function called IORESULT which returns an integer value reporting on the success of the most recent IO transaction. A result of zero indicates a successful transaction; other values are given in the Error Message appendix.

Method 1. This method is the preferred one. Compile the program or module with $SYSPROG$ enabled, and use the TRY-RECOVER statement to trap the errors.

```
$sysprog$
program trapmethod (input,output);
var
  name: string[80];
  f: text;
  ior: integer;
begin
  repeat
    write('Open what file ? ');
    readln(name);
    try
      reset(f,name+',text');
      ior := 0;    (*if we get here, it didn't fail*)
    recover
      if escapecode = -10 then  (*it's an IO error*)
        begin
          ior := ioresult;  (*save it; will be affected by write stmt*)
          writeln('  Can''t open it.  IOresult =',ior);
        end
      else
        escape(escapecode);
until ior = 0;
end.
```

In this case, if the file RESET fails, then the system will set ESCAPECODE to -10 and then execute the RECOVER block. IORESULT will then return the actual I/O error code. Note that IORESULT reports on the last I/O operation attempted; therefore, a variable (ior) was used to store the value before executing the WRITELN statement.

Method 2. This method is used in UCSD Pascal programs. For it to work, you must also suppress the error checks normally emitted by the Compiler.

```
$ucsd$
program ucsdmethod (input,output);
var
  name: string[80];
  f: text;
  ior: integer;
begin
  repeat
    write('Open what file ? ');
    readln(name);
    $iocheck off$
    reset(f,name+'.text');
    $iocheck on$
    ior := ioresult;     (*save it; will be affected by write stmt*)
    if ior <> 0 then
    writeln(' Can''t open it.  IOresult =',ior);
  until ior = 0;
end.
```

Note that $IOCHECK OFF$ before the RESET statement inhibits the setting of ESCAPECODE during this statement. IORESULT will still be set correctly, however.

Heap Management

The "heap" is the area of memory from which so-called dynamic variables are allocated by the standard procedure NEW. When a program begins running, it has available one area of memory for data. The program's stack begins at the high-address end of this area and grows downward; the heap begins at the low-address end and grows upward. If the stack and heap collide, a Stack Overflow error (escapecode −2) is reported.

Two disciplines are available for the recovery of heap variables after they become unwanted: the MARK/RELEASE method, and the DISPOSE method. The first is simpler and faster, the second more general.

MARK and RELEASE

This method uses two standard procedures to manage the heap in a purely stack-like fashion. MARK is called to set a pointer to the next available byte at the top of the heap. Subsequent calls to NEW will all take space from above this point. When the program finishes with all the variables above the mark, RELEASE is called to move the top of the heap (the next available space) back to the value saved by MARK.

```
program markrelease;
type
  ptr = ^ rec;
  rec = record
          f1,f2: integer;
        end;
var
  top,p: ptr;
  i: integer;
begin
  mark(top);          (* remember the base of the heap *)
  repeat
    for i := 1 to 5000 do
      begin
        new(p);       (* allocate from next highest heap address *)
        ...
      end;
    release(top);     (* cut back the heap; recover all space *)
  until false;        (* program will run forever *)
end.
```

When using this method, the computer does not prevent you from making the mistake of releasing to a point **above** the current top-of-heap!

DISPOSE

Alternatively, the standard procedure DISPOSE can be used to return each unwanted dynamic variable back to a pool of free space.

Calls to DISPOSE will have no effect (the freed storage will not be reused) **unless** the main program and the modules containing the NEW and DISPOSE calls are compiled with the directive $HEAP_DISPOSE ON$.

```
program disposal;
type
  ptr = ^ rec;
  rec = record
          next: ptr;
          f1,f2: integer;
        end;
var
  top,p,root: ptr;
  i: integer;
begin
  mark(top);        (* remember the base of the heap *)
  repeat
    root := nil;
    for i := 1 to 5000 do
      begin
        new(p);     (* after disposes, will allocate from free list *)
        p^.next := root;  root := p;  (* chain all cells together *)
        ...
      end;
    ...
    repeat          (* give back all cells one at a time *)
      p := root;
    root := root^.next;     (* follow the chain *)
    dispose(p);  (* mem manager puts on a free list *)
    until root = nil;
  until false;      (* program will run forever *)
end.
```

The recycling algorithm takes advantage of the fact that programs which use the heap operate on a great many variables of just a few types. Each type has a characteristic size. When a variable is disposed, it is saved at the front of a list of other variables of the same size. When a variable is allocated, the NEW routine first looks on the list corresponding to the size required; if there is a free object there, it can be allocated immediately. Usually there will be very little computational overhead for either NEW or DISPOSE.

The memory manager maintains free lists for objects of sizes 4, 6, 8, ... 32 bytes, and one more list for all larger objects. Objects are allocated from this last list on a first-fit basis. No dynamic variable is ever allocated an odd number of bytes.

It is possible for the program to behave so that the heap becomes fragmented (broken into many small pieces). If a request then arrives to allocate space for a large variable, the memory manager will try to recombine the fragments to make a piece big enough to satisfy the request. The fragments must be sorted by address and adjacent ones merged.

The recombination process takes much longer than a simple allocation. Consequently, in real-time applications it is important to analyze the dynamic behavior of programs which use DISPOSE.

Mixing DISPOSE and RELEASE

It is also possible to mix the disciplines in a well-behaved manner. However, not all implementations of HP Pascal allow mixing these methods in a program. A program which does so may not run properly on other implementations.

If you RELEASE a properly MARKed pointer after some calls to DISPOSE, the memory manager will leave on the free lists all disposed objects whose addresses are below the released location. All the space above the released location becomes free, whether or not it was disposed.

During this process the memory manager also recombines any adjacent free fragments, so RELEASE can also be used to reduce fragmentation. Just MARK the current top of the heap, then immediately RELEASE to the same spot.

Converting UCSD Pascal Programs

("UCSD Pascal" is a trademark of the Regents of the University of California)

There are many slightly differing implementations of the UCSD Pascal system. Series 200 Pascal will not provide perfect compatibility with UCSD Pascal or IEM Pascal (HP 9835/9845 systems). In particular, it isn't possible to directly interpret P-code programs since Series 200 Pascal translates programs directly into native MC68000 processor instructions.

Instead the aim is to make it easy to recompile USCD programs to run on your system. We are unable to provide perfect compatibility for several reasons:

1. Technical difficulty in the case of two features
2. Low return / low utilization of some other features
3. Definition conflicts between UCSD Pascal and the HP Pascal standard

Most programs should port easily, but some programmer attention will be required.

This Appendix consists of two parts: a detailed list of those UCSD language extensions which are enable by the $UCSD$ directive, and some notes giving more guidance to the programmer.

Supported Features of UCSD Pascal

To use these language extensions, precede the source program text with the $UCSD$ directive.

UCSD Feature	Series 200 Support
TYPE S = STRING [maxlength]	support
default string length 80	-unsupported; must specify length
string lengths of up to 255 characters	support
arbitrary string type as VAR parameter	-must specify STRING rather than string typename
setting length of string	support
LENGTH of string function	support
POS string position function	support
CONCAT string function	support
COPY substring function	support
DELETE substring function	support
INSERT substring function	support
SCAN string/character array procedure	support
MOVELEFT byte oriented data moving	-support; behavior with overlapped source and destination buffers may not be consistent with other UCSD implementations.
MOVERIGHT byte oriented data moving	-support; behavior with overlapped source and destination buffers may not be consistent with other UCSD implementations.
FILLCHAR byte stream fill	support

UCSD	Series 200 Support
untyped files	support
UNITREAD direct I/O	support
UNITWRITE direct I/O	support
UNITBUSY I/O test	support
UNITCLEAR I/O flush	support
UNITWAIT I/O idle	support
BLOCKREAD direct I/O	support
BLOCKWRITE direct I/O	support
IORESULT function	support
Standard units PRINTER, CONSOLE, SYSTERM	support
SEEK random access positioning	-support but index from one not zero
CLOSE file options LOCK, NORMAL, PURGE, CRUNCH	-the option name must be enclosed in quotes
INTERACTIVE text files	-specifier disallowed but behavior provided by HP TEXT files
SIZEOF variable or type	support
HALT program termination	support
GOTOXY cursor positioning	support
program heading without listing standard files	support
EXTERNAL procedures/functions	support
MEMAVAIL heapspace interrogation function	-returns size in bytes, not words
SETs with up to 4k elements	-support limited to 255 elements
16-bit integer	-normal integer 32 bits; may declare subrange -32768..32767
long BCD integer to 36 digits	-unsupported; have binary 9-and-a-half digit integer.
STR long int to string conversion	-HP Pascal has more general STRWRITE
32-bit real numbers	-9826 always uses 64-bit reals
LOG function	-only LN is supported
TIME function	-not supported; read system clock
PWROFTEN function	not supported
multiword comparison of arrays, records	not supported
nested comments	-use $IF...$ directive instead
CASE statement for illegal selector	-must add OTHERWISE clause

UCSD	Series 200 Support
EXIT statement	-unsupported; can be simulated by TRY/RECOVER (cf. Appendix C)
SEGMENT procedures	-not supported; entire program must be resident.
UNIT	-functionally a subset of MODULE; MODULE syntax a little different
Compiler directives/options	-syntax differs: see Appendix B
AUTOPAGE	-use LINES 2000000 to turn off
COPYRIGHT	COPYRIGHT
DEBUG	DEBUG
FLIP (byteflip)	unsupported (irrelevant)
GOTO	unsupported (always allowed)
IOCHECK	IOCHECK
INCLUDE	INCLUDE
(intermixed declarations in INCLUDE)	supported
LIBRARY	SEARCH
LINESPERPAGE	LINES
LINEWIDTH	PAGEWIDTH
LIST	LIST
LIST <filename>	LIST file specification
LISTFILE	LIST file specification
PAGE	PAGE
QUIET	unsupported (irrelevant)
RANGE	RANGE
SWAP	unsupported
TABLE	TABLES
TRACE	DEBUG and use debugger
TRACEPAUSE	DEBUG and use debugger
USERMODE	unsupported (irrelevant)

Some Useful Hints

Files

UCSD Pascal doesn't prevent writing to a file which was opened for reading (using RESET). The converse is also true. If you get IO error 24, 25 or 26, the file should have been opened using the HP Pascal standard procedure OPEN.

UCSD Pascal's random access mechanism (SEEK) considers that the first component of a file is number zero. HP Pascal considers that files begin with component number one. The $UCSD$ directive does not fix this problem.

UCSD Pascal recognizes a text file type called INTERACTIVE, which differs from files of type TEXT in that a component of the file isn't fetched until it is needed. All HP Pascal text files exhibit this "lazy IO" behavior, so you should change INTERACTIVE files to files of type TEXT.

Strings

In UCSD Pascal, the declaration `var s: string` is equivalent to `var s: string[80]`. HP Pascal requires the length specifier.

A similar comment applies to string value parameters; the specifier "string" is equivalent to the name of an 80-character string type, whereas HP Pascal requires an explicit string typename specifier for value parameters.

UCSD Pascal considers that all strings are compatible as VAR parameters, even if the actual parameter is shorter than the specified formal parameter. This can lead to unexpected bugs. HP Pascal allows two forms of VAR string parameter. If a string typename is used, only another string of identical type may be passed. If the specifier STRING is used, any string may be passed. In the latter case, however, an "invisible" second parameter is also passed, giving the maximum length of the actual parameter. Thus range checking can be performed.

```
program UCSDstrings;
type
  string15 = string[15];

var
  s1: string;
  s2: string [15];
  s3: string[80];

  procedure p1 (s: string);
    ...


  procedure p2 (s: string15);
    ...
  procedure p3 (var s: string);
    ...
  procedure p4 (var s: string15);
    ...


begin
  p1(s1);   {legal}
  p2(s1);   {legal}
  p3(s1):   {legal}
  p3(s2);   {legal}
  p4(s1);   {legal}
  p4(s2);   {legal}


end.
```

```
program HPstrings;
type
  string15 = string[15];
  string80 = string[80];
var
  s1: string80;
  s2: string15;
  s3: string[80];

  procedure p1 (s: string80);
    ...
  procedure p1b (s: string);  {illegal}
    ...
  procedure p2 (s: string15);
    ...
  procedure p3 (var s: string);
    ...
  procedure p4 (var s: string15);
    ...
  procedure p5 (var s: string80);
    ...
begin
  p1(s1);   {legal}
  p2(s1);   {legal}
  p3(s1);   {legal}
  p3(s2);   {legal}
  p4(s1);   {illegal}
  p4(s2);   {legal}
  p5(s1);   {legal}
  p5(s3);   {illegal}
end.
```

The Exit Procedure

In UCSD Pascal, the statement EXIT(proc) causes normal program flow to be altered. The current procedure is discontinued, and procedures are exited in order (most recently called first) until procedure "proc" is exited. The program continues at the next statement after the call on proc.

This Pascal implementation has no exactly comparable feature; the program must be altered. If the EXIT statement occurs within the procedure which is to be exited, a simple goto statement will suffice. Otherwise you must use the TRY-RECOVER statement, which is enabled by $SYSPROG$.

The basic technique is to surround with a TRY the entire body of any procedure which is the target of an EXIT. The EXIT itself is simulated by calling ESCAPE with an error code corresponding to the name of the procedure to be exited. The target procedure catches this escape in its recovery part and then exits normally.

```
program UCSDexits;

  procedure p1;

  begin
    ...
    exit (p1);
    ...
  end;

  procedure p2;
    procedure p3;
    begin

      ...
      exit(p3);
      ...
      exit(p2);
      ...



    end; {p3}
  begin {p2}

    p3;



  end; {p2}

begin {main}
  p1;
  p2;
end.
```

```
$sysprog$
program HPtryrecover;
const  exitp2 = 100;  exitp3 =101;
  procedure p1;
  label 1;
  begin
    ...
    goto 1;  {simple local exit}
    ...
1: end;

  procedure p2;
    procedure p3;
    begin
      try
        ...
        escape(exitp3);
        ...
        escape(exitp2);
        ...
      recover
        if escapecode <> exitp3 then
           escape(escapecode);
    end; {p3}
  begin {p2}
    try
      p3;
    recover
      if escapecode <> exitp2 then
        escape(escapecode);
  end; {p2}

begin {main}
  p1;
  p2;
end.
```

Nested Comments

Comments in Pascal programs may be delimited by either curly braces or parenthesis-asterisk pairs:

```
{ this is a comment }
(* and so is this *)
```

UCSD Pascal requires that the closing delimiter of a comment be the same "kind" as the opening one. HP Pascal treats the two kinds of opening (and closing) delimiter as synonmyms.

```
(* this is an HP Pascal comment }
(* this is all one { UCSD } comment *)
```

The last example will get a syntax error in HP Pascal because the curly-brace after the word "UCSD" terminates the comment.

The easiest way to get around nested comments in a UCSD Pascal program is to surround the outer comment with conditional compilation directives:

```
$if false$
...   all of the materal inside gets skipped   ...
$end$
```

Case Statements

In UCSD Pascal, if the selector of a CASE statement doesn't match any of the labelled cases, the entire statement is skipped. Series 200 Pascal instead reports error −9, "Case statement range error".

This problem can be avoided by putting an OTHERWISE clause at the end of the case statement:

```
   case i of
1:  writeln('case 1');
2:  writeln('case 2');
otherwise
    writeln('The value of i is ',i:5);
   end;
```

Separate Compilation Units

The syntax of UNITs can readily be changed into an equivalent MODULE for compilation by HP Pascal implementations. The word INTERFACE is removed. The word USES is replaced by IMPORT. And the other declarations in the interface part of the UNIT are preceded by the word EXPORT.

```
unit goodstuff;                      module goodstuff;
interface                              import badstuff,betterstuff;
  uses badstuff,betterstuff;           export
  const                                  const
    ... (constant declarations)            ...
  type                                   type
    ... (type declarations)                ...
  var                                    var
    ... (variable declarations)            ...
  procedure p1 (a,b: integer);           procedure p1 (a,b: integer);
  function f(x): real;                   function f(x): real;
implementation                       implement
  ...                                  ...
end.                                 end.
```

Compatibility Rules

HP Pascal enforces stricter compatibility rules than UCSD Pascal. HP generally requires that types be **identical** or **equivalent** where UCSD will accept mere similarity of form.

```
program UCSDisnotpicky;          program HPispicky;
type                             type
  complex = record                 complex = record
              re,im: real                      re,im: real
            end;                             end;
  polar =    record                polar =    record
              r,theta: real                    r,theta: real
            end;                             end;
                                   roundly = polar;
var                              var
  a: complex;                      a: complex;
  b: polar;                        b: polar;
                                   c: roundly;
begin                            begin
  a := b;  { legal }               a := b;  { illegal }
                                   c := b;  { legal }
end.                             end.
```

How Pascal Programs Use the Stack

This section describes how Pascal programs use the stack to store data, return addresses for procedures, and pointers needed to access variables belonging to nested procedures. The information can be useful when writing assembly language routines, and when debugging at the machine level.

You can also investigate this subject by writing some Pascal test programs and seeing what they produce. The Librarian's Unassemble command is very useful for this. Two Compiler directives also produce valuable information: $DEBUG ON$ correlates the machine code displayed by Unassemble with the original Pascal lines, and $TABLES$ causes the Compiler to print a description of the size and location of each object in the program.

The Pascal Stack

Five types of data can be stored on the stack:

- procedure/function parameters
- return addresses
- local variables
- stack frame pointers
- static links

Two address registers are reserved for stack manipulations:

- A7 - the stack pointer (SP)
- A6 - the stack frame pointer (SF)

The stack grows downward in memory as procedures are called, with A7 always pointing to the base (beginning, lowest address) of the datum on the "top" of the stack. That is, when space is allocated for a procedure which has been called, the area allocated has a lower (more negative) address than the space already allocated for the calling procedure. Space allocated to a procedure is called its **stack frame**.

However, variables extend upward in memory. This simply means that the address of the first element of an array, or the first field of a record, is lower than the address of the second element or field.

Global Variables

Register A5 is reserved as the global base register. A reference to any program or module global variable is always formulated as a displacement from where register A5 points. The maximum size of the global area is 64K bytes (the displacement field size). In practice, not all of this space is available to the program. Some of this area is used for system globals, command interpreter globals, permanently loaded programs and modules, and so forth.

See the Assembler chapter for details on how to reference Pascal global variables from assembly language programs.

Procedure Calls

When one procedure calls another, the caller pushes any parameters to the called procedure on the stack. The parameters are pushed on the stack by first decrementing the stack pointer (A7) an amount equal to the size of the parameter, then storing the parameter where SP now points. (Pushing a byte decrements the stack pointer by two, since it must always have an even value.) The calling procedure executes a JSR instruction which pushes the return address on the stack and jumps to the entry point of the called procedure.

The first instruction executed by the called procedure is a LINK instruction. The LINK instruction format and function is illustrated below:

format:	LINK A6,# −d	
function:	A6 → −(SP)	—push the stack frame pointer onto the stack
	SP → A6	—set the stack frame pointer equal to the stack pointer
	SP-d → SP	—drop the stack by the size(d) of the local variables for the called procedure

If the program is compiled with $STACKCHECK ON$ (which is the default), a TRAP instruction is issued instead of LINK. The trap service routine checks for stack overflow as it adjusts A6 and SP. In this case the size #d is stored in the next word after the TRAP instruction.

The stack frame pointer (A6) is used by the called procedure to reference its local variables. See Figure 6 for an illustration of stack usage for level 1 procedure calls. Level 1 procedures are those declared at the global level of a program or module.

If the called procedure is not at level 1, the calling procedure pushes a pointer to the stack frame of the procedure in which the **called** procedure is declared. This pointer is called the **static link**. It is used by the called procedure to resolve references to intermediate variables -- variables which are neither local to the called procedure, nor globals of the program.

An example might help to clarify the static link. Consider the following program structure (indentation indicates nesting):

```
program main
  procedure p1
     procedure p2
        procedure p3
        procedure p4
```

Assume this calling sequence: main calls p1, calls p2, calls p4. If p4 calls p3 then the static link pushed would be that of procedure p2 (since p4 is declared within p2). If instead p4 were to call p2 then the static link would point to p1 (p2 is nested within p1). See Figure 7 for a detailed example of static links.

The called procedure is responsible for stack cleanup and for effecting the return to the calling procedure. Any parameters, local data, or static links belonging to the called procedure must be removed from the stack before returning to the caller. Once this is complete a return to the calling procedure can be performed.

The stack cleanup is performed in two steps:

Step 1: Restore the stack frame pointer. Use the UNLK instruction to remove local data from the stack.

format:	UNLK A6	
function:	A6 → SP	—set the stack equal to the stack frame pointer
	(SP) + → A6	—load the stack frame pointer from the stack and autoincrement the stack pointer (this leaves the stack pointer pointing to the return address)

Step 2: Restore the stack pointer. This removes the static link and parameters from the stack. After this step, the stack pointer should be as it was before the procedure call.

The called procedure returns to the caller by branching to the return address. If the return address was saved in an address register during stack cleanup then an indirect JMP through the address register is executed. If the return address was left on the stack then an RTS instruction is executed.

format:	RTS	
function:	(SP) + → PC	—set the program counter to the value pointed to by the stack pointer and pop the value off the stack

See Figure 8 for an example of a return from a called procedure.

Function Calls

Function calls differ from procedure calls only in that they return results. The result is usually returned on the stack. It is the responsibility of the calling procedure or function to pop the result off of the stack. This is normally done when the result is referenced.

Parameter Passing Mechanisms

There are two kinds of formal parameters: those passed by reference, and those passed by value.

- reference parameters
  all handled alike
- value parameters:
  a) simple value parameters:
  simple types (integer, char.)
  array and record types <= 4 bytes
  b) copied value parameters:
  reals, and array and record types > 4 bytes

Reference parameters are those which are specified VAR in the procedure heading. They are "passed by reference": the address of the actual parameter is passed to the called procedure or function. This address is used for all references to the parameter. No copying of the parameter is performed.

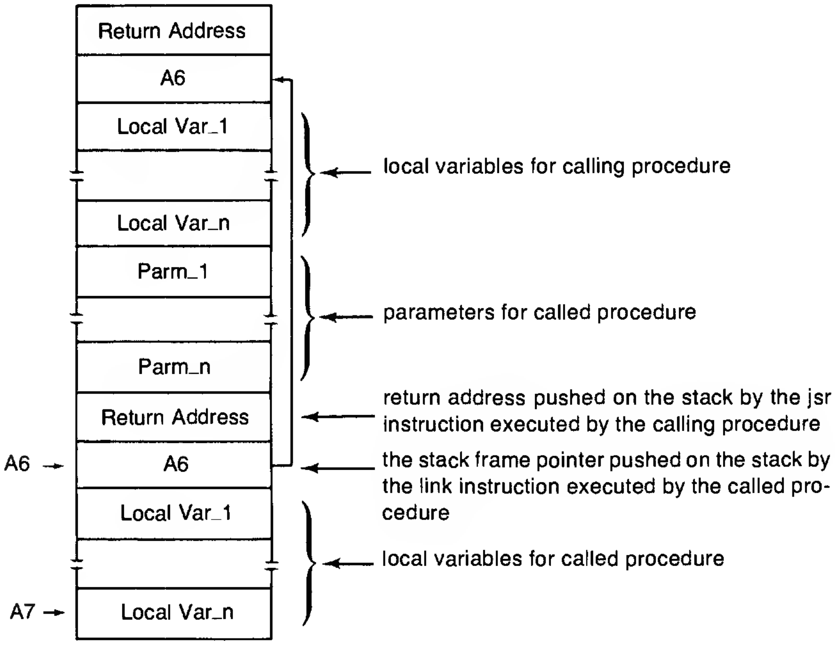
Value parameters are those which are **not** specified VAR in the heading. They are "passed by value": a copy of the parameter is passed to the called procedure or function. If the value parameter is a simple type (except REAL), then its value is pushed on the stack. If the parameter is a simple REAL, or an array or record (and its size is more than 4 bytes), then its address is pushed on the stack by the caller. Before the called routine executes its first statement, it uses the pushed address to copy the parameter into its local data space (the Compiler reserved this space in addition to the local variable space).

Values of type "procedure" are not copied; their values are pushed directly even though they are eight bytes long.

Function Results

Sometimes the calling environment must allocate temporary space in which to return function results. In general this is necessary when the function returns a result which is bigger than 4 bytes. The temporary space is allocated as part of the program's global area if the call is from the main program; otherwise it is allocated as part of the local data area. The amount of temporary space required is determined at compile time. Functions which return a value of type real are an exception; the result area is on the stack and occupies eight bytes.

Figure 6. PASCAL Procedure Calls (Without Static Links)


Return Address
A6
Local Var_1
Local Var_n
local variables for calling procedure
Parm_1
Parm_n
parameters for called procedure
Return Address
return address pushed on the stack by the jsr instruction executed by the calling procedure
A6
A6
the stack frame pointer pushed on the stack by the link instruction executed by the called procedure
Local Var_1
local variables for called procedure
A7
Local Var_n

(The stack is pictured growing toward the bottom of the page. Pointers actually address the bottom of the designated entry.)

The following Pascal program illustrates the use of the static link. Note that the program is only intended for illustration purposes; running it results in error -2 (not enough memory), because it recurses infinitely.

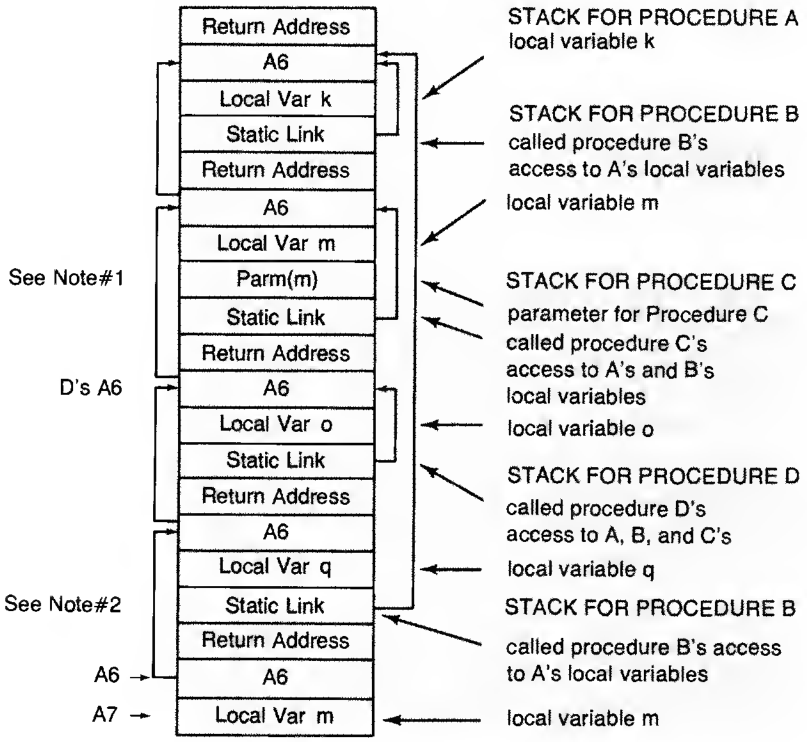
```
program main(input,output);
var  i: integer;

    procedure A;
    var  k:integer;

        procedure B;
        var  m: integer;

            procedure C(i:integer);
            var  o: integer;

                procedure D;
                var  q: integer;
                begin
                  B;
                  C;
                  i := k;
                  k := m; { see note #3 }
                  m := o;
                  o := q;
                  q := 1;
                end; {D}

            begin {C}
              D;
              m := o;
            end; {C}

        begin {B}
          C(m);
          k := 1;
        end; {B}

    begin {A}
      B;
    end; { A}

begin {main}
    A;
end. {main}
```

Consider the following calling sequence:

main calls A
    A calls B
    B calls C (with m as the parameter)
    C calls D
    D calls B

The stack for this calling sequence is shown in Figure 7.

Figure 7. PASCAL Procecure Calls With Static Links



Note 1: The static link and the parameters are always accessed at positive offsets from A6. The effective address of the static link (if present) is always 8(A6). Local variables are at negative displacements from A6.

Note 2: In general the static link gives the called procedure access to the intermediate variables of procedures which precede it in the calling sequence. In this particular case the static link gives procedure B access to variables declared within procedure A.

(Pointers actually address the bottom of the designated entry.)

Procedure D reaches intermediate variables using:

- its current stack frame pointer
- the difference between its nesting level and that of the called procedure

In this case procedure D is at level 4 and procedure B is at level 2, for a relative distance of 2. Therefore procedure B must follow two static links to reach the stack frame of B.

In other words:

```
MOVEA.L 8(A6),A0        - get procedure D's static link
MOVEA.L 8(A0),A0        - get procedure C's static link
MOVE.L 8(A0),-(SP)      - get procedure B's static link
                          and push it on the stack
```

Remember: procedure C's static link gives access to B's locals,
procedure B's static link gives access to A's, etc.

Note 3: When nested procedures reference intermediate variables they use the static link. An example of this is when procedure D references k and m in the statement k : = m;.

k is declared in procedure A and m is in procedure B. The nesting level relative to procedure D for k is 3 and for m it is 2.

The following code will perform the statement k : = m.

```
MOVEA.L 8(A6),A0        get D's static link
MOVEA.L 8(A0),A0        get C's static link
MOVEA.L 8(A0),A0        get B's static link

MOVEA.L 8(A6),a1        get D's static link
MOVEA.L 8(a1),a1        get C's static link

MOVE.L -4(a1), -4(A0) store m in k
```

The return to procedure A (as shown in the following stack segment) is accomplished in four steps. Note: the register prefixes indicate the value of the register for the indicated step. The values are those the registers have AFTER the step has been executed.

```
Step 1: UNLK A6            - restore the stack frame pointer (A6)
Step 2: MOVEA.L (SP)+,A0   - save the return address in A0
Step 3: ADDQ.l #12,SP      - restore the stack pointer
Step 4: JMP (A0)           - return to where procedure A called B
```

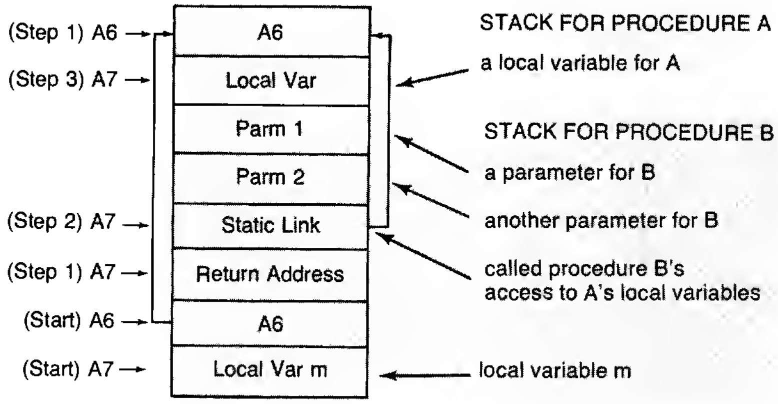
If there are not any parameters then the return sequence normally is:

```
UNLK A6              - restore the stack frame pointer
MOVEA.L (SP)+,(SP)   - replace the static link with the return
                       address
RTS                  - return to procedure A
```

If there is no static link (and no parameters) then the sequence is:

```
UNLK A6              - restore the stack frame pointer
RTS                  - return to procedure A
```

Figure 8. Return from a Procedure Call



(Pointers actually address the bottom of the designated entry.)

Implementation Restrictions

Input to the Compiler

Input to the Compiler is normally prepared by the Editor. Files produced by the Editor are either TEXT, ASCII or DATA. If the file name contains no suffix, TEXT will be assumed.

Nesting of INCLUDE Files and IMPORT Declarations

The Compiler can keep track of a maximum of 10 active input files at once. This means that an INCLUDEd file can include another file, and so forth, nine times. Exceeding this limit causes error 608 or 610.

When a module is imported which hasn't been previously imported during a compilation, a form of inclusion takes place in which various library files are opened and searched. These files are counted against the maximum of ten while they are open (during the processing of the IMPORT declaration).

If module "A" is imported, and its interface specification imports module "B", and so on, the Compiler will chase the importation chain to its very end (unless it runs into the name of a module which has already been seen). If you encounter a situation in which the chain exceeds the limit of ten open input files, you can avoid the problem by making the first module in the chain import all the others in reverse order: the end of the chain first, then the modules which depend on that last one, and so on.

Module Names Used by the Operating System

If you create a module having the same name as a system module, and your module exports a procedure which has the same name as some procedure exported from that Operating System module, the loader may hook up external references to the wrong place. That is, your module will override the Operating System Module.

The way to avoid this is to not use any of the module names listed in the Technical Reference Appendix.

Maximum Size of Local and Global Data Areas

Global areas are accessed by adding (subtracting) a displacement value to the contents of processor register A5. Since the displacement value cannot exceed 32 768 bytes, the A5 register points to 32 768 bytes below the start of the global area, thus providing access to a full 65 536 bytes of global space.

Every module loaded is allocated global area at load time. The sum of global space for all the modules and programs loaded at one time can't exceed 65 536 bytes. About 2 Kbytes of global space is taken up by the system. The table below shows the approximate values of global space required for each subsystem.

	Total	**Globals**	(in Kbytes)
Assembler:	65	7	
Compiler:	195	7	
Editor:	50	4	
Filer:	48	1	
Librarian:	47	2	

Table 1.

Note

The only way to remove a permanently loaded program or library from memory is to reboot.

If you're writing a program which needs a very large global area (**ie** a big array), it can be allocated out of the heap by a call to NEW, then referenced through a pointer. This is a bit of a nuisance, but carries a negligible performance penalty.

```
program bigarray;
type
  gigantic = array [1..20000] of real;   (*160,000 bytes*)
  ptr = ^gigantic;
var
  bigthing: ptr;
  i,j: integer;
begin
  new(bigthing);
  for i := 1 to 20000 do bigthing^[i] := 0.0;
end.
```

Implementation of CASE Statements

CASE statements are implemented using a "jump table". This table is organized as an array of 16-bit values, each an "offset" or distance from the head of the statement to the various cases. The number of entries in the table is the inclusive range from the lowest to the highest labels in the statement. If the lowest is labeled "1" and the highest is "15000", there will be 15000 entries!

The Compiler displays a warning if it decides a CASE statement is unreasonably large and most of the values in the table are absent or correspond to the same case. If you get such a warning, you should probably recode the statement using IF's or a combination of IF's and a smaller CASE statement.

Despite the warning, the Compiler will try to generate the statement as written. If the jump table is very large, it may take a **long** time to write to the output file. You may even think the Compiler has gotten hung up somehow, but the warning message indicates this is not the case.

Range of Real and Integer Numbers

- integers: – 2147483648 thru 2147483647
- reals: – 1.797693134862315 E + 308 thru – 2.225073858507202 E – 308
  0.0
  + 2.225073858507202 E – 308 thru + 1.797693134862315 E + 308
- longreals: same as reals

16-bit Subranges

A variable declared as a subrange needing 16 or fewer bits for its representation will be stored as a word instead of a longword. For example,

```
type integer = -32768..32767;
```

If all the operands of an expression are represented as 16-bit objects, the Compiler implements the expression in 16-bit rather than 32-bit instructions. In particular, integer overflow is detected as a carry into the 17th bit. The rules are:

- add, subtract: overflow will be detected.
- divide: – 32768 div – 1 yields integer overflow.
- multiply: the result is widened to 32 bits.

Note that the representation of an unpacked subrange of integer always reserves room for a sign bit. Hence the range 0..65535 will not be represented in 16 bits, even though it could in fact be.

String Length

Strings may not have a declared length greater than 255 characters.

Ordinal Range of Sets

Sets may not span an ordinal range of more than 256 elements.

Declared Record Length

Records may not be declared which require more than 32767 bytes for their representation.

Deviations from HP Standard Pascal

Known Deviations

The following are known deviations of this implementation from HP Standard Pascal. Although we have tried to identify all deviations, no guarantee is made that this list is complete.

- The standard function LINEPOS is not implemented.
- Type REAL has the same precision as LONGREAL. This is permitted by the Standard. However, in WRITE statements the default field width for LONGREAL is the same as for REAL, and the exponent is written preceded by E instead of L.
- The syntax of the two Compiler options $IF and $SEARCH do not conform to the syntax of all other allowable options.
- Module names are restricted options to a maximum of 15 characters. No other identifiers are restricted in length in this implementation.

File System Differences

To allow for the fact that different computers provide different underlying operating system support, HP Pascal allows certain variations in the parameters passed to the standard procedures for opening and closing files. These parameters appear as strings passed to the standard procedures; it is their content which may vary. For instance, the file naming conventions are very different in different operating systems. Such variations may require minor changes in a program if it is moved to a type of computer different from the one on which it was developed.

Unreported Errors

Certain errors in Pascal programs are not reported by this implementation.

- Disposing a pointer while in the scope of a WITH referencing the variable to which it points.
- Disposing a pointer while the variable it points to is being used as a VAR parameter.
- Disposing an uninitialized or NIL pointer.
- Disposing a pointer to a variant record using the wrong tagfield list.
- Assigment to a FOR-loop control variable while inside the loop.
- GOTO into a conditional or structured statement.
- Exiting a function before a result value has been assigned.
- Changing the tagfield of a dynamic variable to a value other than was specified in the call to NEW.
- Accessing a variant field when the tagfield indicates a different variant.
- Negative field width parameters in a WRITE statement.
- The underscore character "_" is allowed in identifiers. This is permitted in HP Pascal, but is not reported as an error when compiling with $ANSI$ specified.
- Value range error is not always reported when an illegal value is assigned to a variable of type SET.

The Assembler

Chapter 6

Introduction

The Assembler is included in the Pascal Workstation System for translation of assembly language routines into object code. Assembly language programming gives the user the ability to optimize critical sections of a program (such as for speed or code-size improvements).

The Assembler is designed to translate assembly language as specified by Motorola in the *MC68000 User's Manual.* A copy of that manual is included in the Pascal documentation package.

This chapter contains the information necessary to write assembly language programs and subroutines. The first section demonstrates the method of generating external procedures and entire object modules of assembly code that can be interfaced to Pascal programs. The later sections explain the instruction format requirements and the Assembler directives.

The requirements for writing routines that will interface with Pascal programs are explained in detail. You should be familiar with the concept of Pascal modules before attempting to emulate them in assembly language. This is also true of the TRY-RECOVER mechanism. Refer to the Pascal Compiler chapter for this information.

Like Compiler options, the options (directives) that you give to the Assembler are coded into your program and not given in an interactive session. The Assembler directives are explained in the Assembler Pseudo-op Reference section of this chapter.

Operating the Assembler

This section shows you how to:

- Invoke the Assembler
- Specify the name of your text file program and your resulting code file
- Give listing specifications
- Interpret the listing

Invoking the Assembler

The Assembler is delivered on the ASM: disc. If you plan to run the Assembler several times in a session, you could use the Permanent command to keep the Assembler in memory ready to run. Otherwise, put the ASM: disc in a drive and press A to run the Assembler.

Source File Specification

If there is a work file (see the Filer chapter), that file will be automatically assembled and there will be an "errors only" listing on the CRT. If the "errors only" listing is sufficient, your source program file can be specified as the work file. Otherwise, clear the work file.

If there is no work file, you will be prompted to enter the name of your program file:

```
What source file?_
```

Enter the volume name (unless using the default volume explained in the Filer chapter), and file name of your source program. It is not necessary to include the ".TEXT" suffix of your file name. If it is not included, it will be done for you by the system. For example, if your program file is called PROGRAM.TEXT and it is on the volume called TOMS:, then use this file specification:

```
TOMS:PROGRAM
```

Listing File Information

You are then prompted to specify whether or not you will want a listing of the assembly:

```
Do you want a program listing (y/n/e) ?_
```

You may type:

Y for a complete listing
N for no listing but errors reported on the CRT
E for a listing of the errors only

If you want a listing, you can have it printed immediately or have the Assembler generate a file of the listing information:

```
What listing file (default PRINTER:PROGRAM.ASC) ?_
```

For a printer listing, press Return or ENTER.

To generate a listing on a file, enter the name of the volume and the name of the file. It is recommended that a size specification be made for the listing file (See the Filer chapter). Otherwise, the largest space on the disc will be reserved for the listing, which may leave no space for the code file. A good rule of thumb is to use twice the number of blocks used by program file. For example, if TOMS:PROGRAM.TEXT is 20 blocks long, a size specification of 40 blocks is made for the listing file.

```
TOMS:PROGLIST.TEXT[40]
```

(Be sure to include the period at the end of the file name.)

It is possible to have a CRT screen listing by specifying "CONSOLE:" as the listing file. This is not recommended unless the program is very small, or an "error only" listing was requested. The listing will be scrolled onto the screen and you are returned to the Main Command Level. There is no way to control the screen listing.

Object File Specification

Finally, you are prompted to give a name for the code file that will be generated by the Assembler. The default name is that of the source file with the suffix: ".CODE" replacing ".TEXT".

```
Output file (default is TOMS:PROGRAM.CODE) ?_
```

If the default name is acceptable, press (Return) or (ENTER). If you want to specify another name, enter the complete file specification.

After this entry, the Assembler begins processing your program. The CRT displays when the first pass of the Assembler is completed along with the number of errors encountered during the pass. There is a similar display for the second pass. After the second pass is completed, you are returned to the Main Command Level. If no errors were generated during the assembler, a code file was created.

If the assembly program is executable (has a start address), you may run it by pressing (R) at the Main Level. The Run command will run your program automatically until:

- another program is assembled or compiled.
- a workfile is specified.
- the computer is powered down.
- the system volume is re-specified.

If the Run command no longer works for your program, use the eXecute command and give the name of the code file that was generated.

Interpreting the Listing

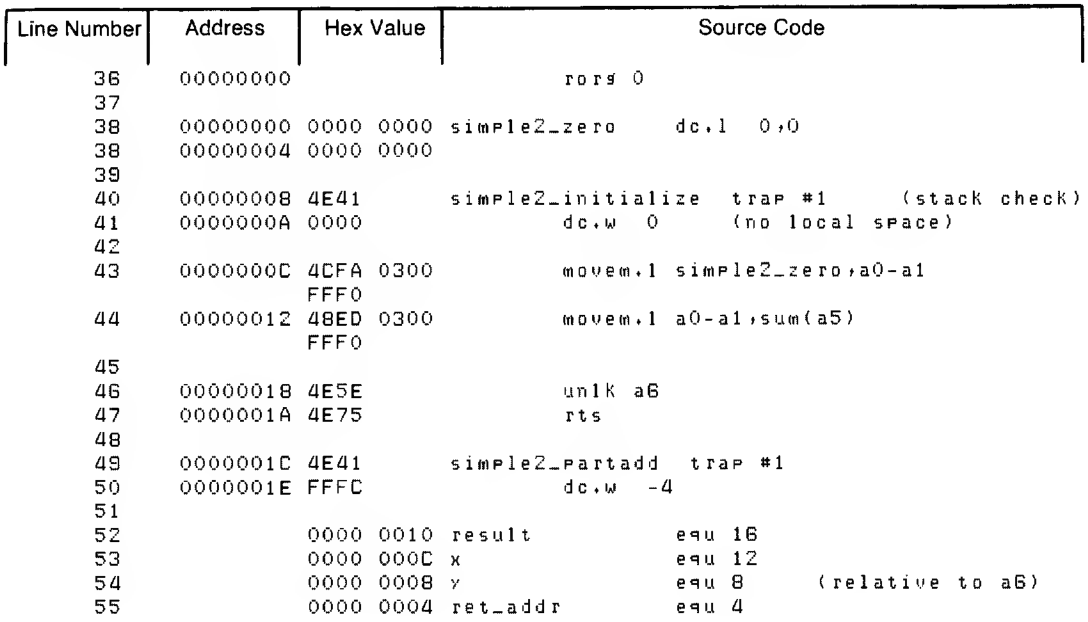
The output from the Assembler contains the following information. The first column on the listing indicates the (decimal) number of the source-program line. For each line of source, a line number is generated. This is true of blank lines as well.

The second column shows the location counter (relative to the code origin). The value is in hex notation unless the DECIMAL pseudo-op is specified. When the program is loaded, the number in column two can be added to the base address of the load to obtain the absolute address of the instruction. This is useful information when debugging.

The third column shows the hex representation of the machine code that is generated by the Assembler for the instruction.

The right side of the listing is a copy of the source program.

Sample Assembler Output



```
Line Number   Address    Hex Value                      Source Code

      36      00000000                        rorg 0
      37
      38      00000000 0000 0000 simple2_zero     dc.l   0,0
      38      00000004 0000 0000
      39
      40      00000008 4E41      simple2_initialize  trap #1      (stack check)
      41      0000000A 0000               dc.w  0      (no local space)
      42
      43      0000000C 4CFA 0300          movem.l simple2_zero,a0-a1
                       FFF0
      44      00000012 48ED 0300          movem.l a0-a1,sum(a5)
                       FFF0
      45
      46      00000018 4E5E               unlk a6
      47      0000001A 4E75               rts
      48
      49      0000001C 4E41      simple2_partadd  trap #1
      50      0000001E FFFC               dc.w  -4
      51
      52               0000 0010 result           equ 16
      53               0000 000C x                equ 12
      54               0000 0008 y                equ 8      (relative to a6)
      55               0000 0004 ret_addr         equ 4
```

Error messages are listed under the line in which they occur. At the completion of the assembly, the number of errors will be displayed. If there are errors, there will be a directive for you to check the location of the last error in the program. At that location there will be a description of the error. Also listed will be the location of the error above it if one exists. In this manner, all errors can be located without having to search the whole listing.

The Programming System

It is assumed that you will be writing most of your programs in Pascal. In the instance where the execution speed of a particular routine is insufficient, this section will show you how to translate the Pascal routine into an assembly language routine and call it from your Pascal program.

It is possible to write a simple procedure, put it in the system library (usually a file named LIBRARY on the * volume), and access it with an EXTERNAL declaration from the Pascal program. However, add some interface text to the routine, and you have created a module. The benefits of modules are that global variables and constants may be used for communication among modules. Special types which define parameters need only be declared in the module containing the called procedure.

A Pascal module was developed for use as an example. The Librarian was used to disassemble the code into its assembly language counterpart. The intent of this section is to explain the method of interpreting the disassembly information and producing a working Assembler language module. The listings of the examples are included at the end of this chapter. The examples are also available on the documentation disc (DOC:). The file (ASMB_P1) imports the file (ASMB_M1). These are both Pascal files. The Pascal file (ASMB_P2) imports the Assembler language file (ASMB_M2).

You'll notice in the example program that the variables are declared to be of the type which are defined in the imported module. If the program merely declared one or two of the procedures to be EXTERNAL procedures, those special types would have to be defined in every program that called the procedures. It would be like going to the Library for a book and having to write down the table of contents every time you wanted to use the book.

For your Assembler language module to interface cleanly with the Pascal program, the conventions of the Compiler must be followed. That is, you must set up the Assembler language module to act as if it were a compiled Pascal module. You must also exit the module leaving everything in order, as a Pascal module would.

The information you need to accomplish a clean "Pascal-to-Assembler language" interface is presented in this section. You should understand how the Compiler:

- Prepares interface text (IMPORT text)
- Declares entry points (DEF table)
- Declares external references (EXT table)
- Passes parameters
- Creates global variable space
- Initializes modules
- Recovers from errors
- Returns from subroutines

You will find a listing of the Pascal program and module as originally written, a listing of the disassembly of the module, and a listing of the final, working Assembler language module. These listings are included at the end of this section. It might be helpful to remove them from the manual and keep them out for reference as you're reading this material.

The first subject covered is the method of generating the IMPORT text. This is what separates an importable module from a simple external routine. The subsequent material is of concern in either case. There will be a short explanation of the method for declaring EXTERNAL routines toward the end of the section.

The IMPORT Text

Certain information must be passed from an imported module to the Compiler to complete the module interface. This information is the IMPORT text. Actually, IMPORT text contains IMPORT declarations and EXPORT declarations. It's called IMPORT text because it's what the Compiler needs when it is importing the module. It must know the module name, global variables, global constants, and procedure and function names. If special TYPE declarations are needed to define the variables, they must be included in this information.

At compile time, your imported Assembler module must make this information available to the Compiler. This is done with the SRC pseudo-op. See how the IMPORT text of the Pascal listing is exactly the same as the SRC-IMPORT text below.

```
src module simple2;
src export
src   type
src      rec = record
src         i1 : integer;
src         i2 : integer;
src      end;
src   const
src      zero = rec[i1:0,i2:0];
src   var
src      lastresult : rec;
src   procedure initialize;
src   procedure add(a,b : rec;
src                 var out : rec);
src end;
```

The SRC section does not actually name the module or get the global space. There are separate techniques for accomplishing these things, which are discussed later.

The DEF Table

The DEF table contains the locations of all the entry points in the Pascal module and the location of its global space. This information is provided for the linking loader. The information is used to link all the modules together before they can be loaded and executed.

```
DEF table of 'SIMPLE':

   SIMPLE                         Gbase
   SIMPLE_ADD                     Rbase+82
   SIMPLE_INITIALIZE              Rbase+10
   SIMPLE_SIMPLE                  Rbase+252
   SIMPLE_ZERO                    Rbase
```

The symbol "SIMPLE" which is the same as the module name, is the name of the module's global variable space. This symbol is entered into the DEF table automatically when you reserve the global space using the COM statement. This is explained later in the global variable section of this chapter.

"SIMPLE_ADD" AND "SIMPLE_INITIALIZE" are the entry points into the two procedures "add" and "initialize". When writing Assembler language routines, they must be named as the Compiler names its procedures. The Compiler appends the module name to the front of the procedure name, separated by an "_". When the Compiler looks at your IMPORT section, it assumes that the procedures have been named by its convention. When it's time for the loader to hook everything together, it looks for those procedure names in your module's DEF table.

"SIMPLE_SIMPLE" is the entry point, or location, of the module initialization body. Module initialization is discussed later in this chapter.

"SIMPLE_ZERO" is the location of the structured constant, "zero", which appears in the IMPORT section of the module. Any code which resides in the assembly module and is declared in the IMPORT section of the module, must appear in the DEF table. It, too, must be named by prefixing the module name to the constant name that you declare in the IMPORT section. This name must appear as a label at the constant's location in the program.

You must create a DEF table for the Assembler version of your routine. This is done using the DEF statement. Notice that all the symbols in the Pascal module's DEF table are named in the DEF statements below except the symbol for the global variable space. The global variable symbol is entered into the table at the time the space is reserved with the COM statement.

```
def simple2_add
def simple2_initialize
def simple2_zero,simple2_simple2
```

The EXT Table

The EXT table that you get from the Librarian is the list of the symbols that the loader must find in some corresponding DEF table so our module can access those external items.

```
EXT table of 'SIMPLE':

   SYSGLOBALS
```

"SYSGLOBALS" is the only symbol in this particular list. We need to access some of the system's global variables in our routine so we must know where they are kept. They are in the global variable space for the system, "SYSGLOBALS". (See the TRY-RECOVER section for more details about the system globals.)

The EXT table is created in the Assembler module using REFA and REFR. Both instructions enter symbol names into the EXT table. REFA causes the symbol to be referenced using absolute addressing. REFR causes the symbol to be referenced using 16-bit PC relative addressing. See REFA, REFR, SMODE and LMODE in the pseudo-op reference section.

In the example, "SYSGLOBALS" was declared as external using REFA.

If other modules' global variable sections were to be referenced, the symbol for those areas would also need to be included in our EXT table. This is explained in the global variable section.

Declaring the Module Name

The module is named using MNAME. This puts the name of the module in the module directory for the Compiler to reference when importing the module.

If no MNAME is used, the module name will be the same as the file name.

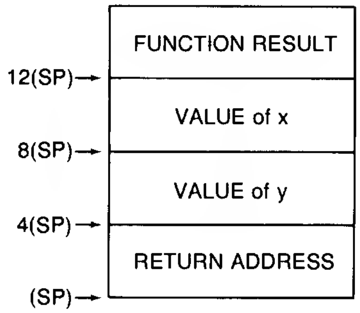
Passing Parameters

When parameters are passed to a procedure, the values or addresses of variables in the parameter list are pushed onto the stack. The function result space is put on the stack if the routine is a function. The leftmost variable in the parameter list is pushed onto the stack. Then the rest are pushed onto the stack in order from left to right. The return address is pushed onto the stack automatically by the processor at the time the JSR instruction is encountered.

For example:

```
114   2F0E              move.l a6,-(sp)
116   487A 005A         pea Rbase+208
120   2B4F FFF6         move.l sp,SYSGLOBALS-10(a5)
124   598F              subq.l #4,sp
126   2F2E FFF0         move.l -16(a6),-(sp)
130   2F2E FFF8         move.l -8(a6),-(sp)
134   4EBA FF98         jsr Rbase+32
138   2B5F FFF8         move.l (sp)+,Gbase-8(a5)
142   598F              subq.l #4,sp
144   2F2E FFF4         move.l -12(a6),-(sp)
148   2F2E FFFC         move.l -4(a6),-(sp)
152   4EBA FF86         jsr Rbase+32
156   2B5F FFFC         move.l (sp)+,Gbase-4(a5)
160   202D FFF8         move.l Gbase-8(a5),d0
164   D1AD FFF0         add.l d0,Gbase-16(a5)
```

The stack is mapped in the following way:



Notice that the stack grows downward (toward smaller addresses).

If a parameter is passed by reference, a 4-byte address is pushed onto the stack. When passing by value, values up to 4-bytes are pushed onto the stack, but larger values are essentially passed by reference. That is, a 4-byte address is pushed on the stack. In this case, a copy of the value must be made in local variable space so that the actual parameter is not altered. This is illustrated in the Local Variable section.

More information can be found in the Compiler chapter under the heading "How Pascal Uses the Stack".

Declaring Global Variables

You must understand how the Compiler allocates global variable space so that you get and use global space the same way. The value stored in register A5 is the base address for all global areas. Each module that declares global variables is allocated an area for them. The symbol assigned to the area is the distance from the base address in A5 to the top of the global area. Globals are then referenced symbolically, using the global area name and offset relative to A5.

The name for the location of a module's globals (relative to the address in A5) is the same as the module name. So the symbol for the global area for "module simple" would be "SIMPLE".

Determine how much space you need for your globals. When determining how much space is needed, you must also consider any variables that are internally global to the module. Notice on the Pascal module listing that the variable "sum" is global to the module.

If you are rewriting a Pascal module as we have done in the example, the Compiler provides variable size information beside the variable declarations on the listing (the negative number). More detailed information can be displayed using the Compiler's $TABLES$ directive. In Assembler language modules, you must also specify the size as a negative value. Declare global space using the COM statement:

```
COM simple2,-16
```

The value, −16, corresponds to the total size of global variables "lastresult" and "sum". Both are records containing two integers each.

The COM statement also enters the symbol into the DEF table.

Referencing Global Variables

The Assembler module name is SIMPLE2, as is its global base. Notice in the DEF Table that "SIMPLE" is equal to "Gbase" (Global BASE) for the Pascal module. Global locations in the disassembly of the Pascal module are referenced using the symbol "Gbase" rather than "simple".

```
DEF table of 'SIMPLE':

   SIMPLE                              Gbase
   SIMPLE_ADD                          Rbase+82
   SIMPLE_INITIALIZE                   Rbase+10
   SIMPLE_SIMPLE                       Rbase+252
   SIMPLE_ZERO                         Rbase
- - - - - - - - - - - - - - - - - - - - - - - - - -
   170    202D FFFC          move.I Gbase-4(a5),d0
   174    D1AD FFF4          add.l d0,Gbase-12(a5)
   178    4E76               trapv
   180    206E 0008          movea.l 8(a6),a0
   184    4CAD 1E00          movem.w Gbase-8(a5),a1-a4
```

When writing your Assembler language module, use the COM symbol to reference globals. The Assembler doesn't recognize "Gbase". In our Assembler module, the global variables are referenced using "SIMPLE2".

```
lastresult       equ simple2-8
lastresult_i1    equ simple2-8
lastresult_i2    equ simple2-4
sum              equ simple2-16  (all are relative to a5)
sum_i1           equ simple2-16
sum_i2           equ simple2-12
escapecode       equ sysglobals-2
recover_rec      equ sysglobals-10
```

Note

When structured variables are used, the individual elements of the structure are referenced at progressively higher addresses within the structure's space.

If, for example, you had declared two integers separately instead of together in one record, you would refer to them as:

```
lastresult_i1        EQU    simple2-4
lastresult_i2        EQU    simple2-8
```

Referencing Other Module's Globals

When referencing the global variables of another module, it is necessary to establish the external reference using REFR or REFA.

The individual variables are referenced at negative offsets from the symbol and relative to A5, as described in the global variable section above. As was mentioned previously, offsets into data areas are provided on Compiler listings.

Local Variables

There are several methods for getting local variable space. The following method is recommended for those intending to produce purely relocatable code. This is important if the code is to be committed to ROM.

Notice that the first instruction in each of the disassembled routines is:

```
TRAP #1
```

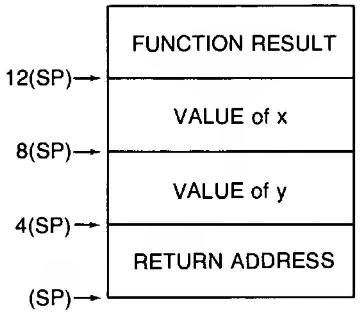
TRAP #1 calls a system routine which allocates local variable space in a new stack frame. A check is made of available stack space. If there isn't room on the stack, a "Not Enough Memory" error is reported and control is transferred to the Main Command Level.

The TRAP #1 routine then executes a LINK instruction. The LINK instruction is explained in detail in the *MC68000 User's Manual* and in the Compiler chapter under "How Pascal Uses the Stack".

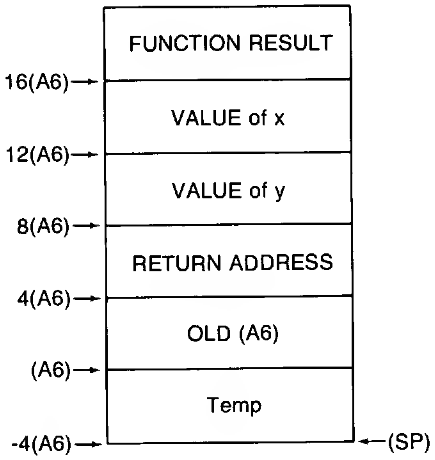
Our Assembler does not understand the double operand format of the TRAP instruction as it is printed in the disassembly listing. The size of the stack frame is specified following the TRAP instruction in a DC.W instruction. The value of the constant in the DC.W instruction specifies the amount of space needed for local variables.

The following illustration shows the stack before the function "part_add" gets its local variable space.

Before the LINK:

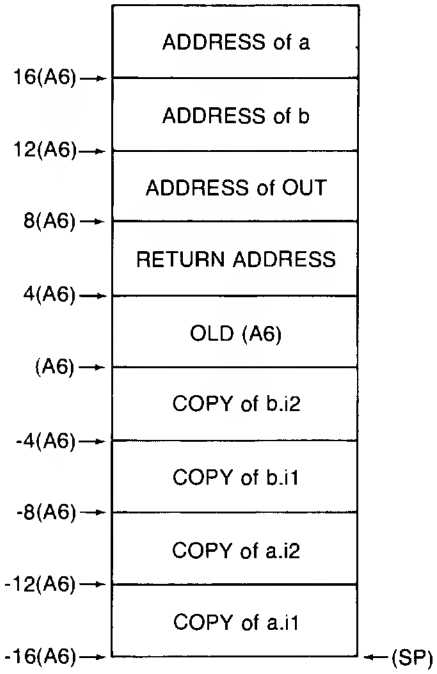


After the LINK:



Parameters are now referenced relative to A6 instead of SP. Local variables are referenced at negative offsets from A6.

Local variable space is also needed for copies of some value parameters. As was discussed in the parameter section, value parameters which are larger than 4 bytes have their address put on the stack in place of the value. In order not to alter the value of the actual parameter, a copy must be made in local variable space. Allocate the space using the TRAP instruction, then immediately move the values of the value parameters into the local variable space. This is the case with the parameters to "Procedure Add".



This mapping was accomplished by the following block of code:

```
      76    504F               addq.w #8,sp
      78    4ED0               jmp (a0)
      80    0000               dc.w 0          or dc.b 0,0        or dc.b '  '
- - - - - - - - - - - - - - - - - - - - - - - - - - - - SIMPLE_ADD
      82    4E41 FFF0          trap #1,#-16
      86    206E 0010          movea.l 16(a6),a0
      90    2D58 FFF0          move.l (a0)+,-16(a6)
      94    2D50 FFF4          move.l (a0),-12(a6)
      98    206E 000C          movea.l 12(a6),a0
     102    2D58 FFF8          move.l (a0)+,-8(a6)
     106    2D50 FFFC          move.l (a0),-4(a6)
     110    2F2D FFF6          move.l SYSGLOBALS-10(a5),-(sp)
     114    2F0E               move.l a6,-(sp)
     116    487A 005A          pea Rbase+208
     120    2B4F FFF6          move.l sp,SYSGLOBALS-10(a5)
```

Module Initialization

Finally, it is necessary to include a module initialization body within each module. The initialization body is a routine which is named by appending the module name to itself, separated by “_”.

The purpose of module initialization is to allow for file initialization within the module. Even if a module declares no files, the Compiler emits a call to the module initialization body for every module imported into a program. It can be a null routine such as an RTS with the label tacked on to the end of the assembly:

```
simple2_simple2     rts
```

The name of the module initialization body must be marked as an entry point along with the other procedure names using DEF.

Error Recovery

The TRY-RECOVER escape mechanism can be written into assembly language routines for graceful termination of programs that generate errors. TRY-RECOVER is explained in detail in the Compiler chapter under the heading “System Programming Language Extensions”.

The section of code that could cause the error is enclosed within the TRY section. The TRY section creates a RECOVER-record on the stack. The record contains the location of the previous RECOVER-record, the stack frame pointer, (A6), and the location of the RECOVER code. The location of this record is saved in a special location that the system knows about. This location is at an offset of −10 in “SYSGLOBALS” (operating SYStem GLOBALS). “SYSGLOBALS” is relative to A5.

An example of the TRY action is taken from the disassembly:

```
 86   206E 0010        movea.l 16(a6),a0
 90   2D58 FFF0        move.l (a0)+,-16(a6)
 94   2D50 FFF4        move.l (a0),-12(a6)
 98   206E 000C        movea.l 12(a6),a0
102   2D58 FFF8        move.l (a0)+,-8(a6)
106   2D50 FFFC        move.l (a0),-4(a6)
110   2F2D FFF6        move.l SYSGLOBALS-10(a5),-(sp)
114   2F0E             move.l a6,-(sp)
116   487A 005A        pea Rbase+208
120   2B4F FFF6        move.l sp,SYSGLOBALS-10(a5)
124   598F             subq.l #4,sp
126   2F2E FFF0        move.l -16(a6),-(sp)
130   2F2E FFF8        move.l -8(a6),-(sp)
134   4EBA FF98        jsr Rbase+32
138   2B5F FFF8        move.l (sp)+,Gbase-8(a5)
```

After the above code has been written, write the code body of the routine.

The last piece of code must restore the pointer to the previous RECOVER-record and remove the current one from the stack. Control is then transferred to the instruction following the RECOVER section. For example:

```
178    4E76              trapv
180    206E 0008         movea.l 8(a6),a0
184    4CAD 1E00         movem.w Gbase-8(a5),a1-a4
       FFF8
190    4890 1E00         movem.w a1-a4,(a0)
194    2B6F 0008         move.l 8(sp),SYSGLOBALS-10(a5)
       FFF6
200    DEFC 000C         adda.w #12,sp
204    4EFA 0024         jmp Rbase+242
208    2C5F              movea.l (sp)+,a6
210    2B5F FFF6         move.l (sp)+,SYSGLOBALS-10(a5)
214    7064              moveq #100,d0
216    B06D FFFE         cmp.w SYSGLOBALS-2(a5),d0
220    6600 0012         bne Rbase+240
224    4CBA 0F00         movem.w Rbase,a0-a3
```

If an error or exception does occur, the system stores the number of the error in a location at "Sysglobals-2(A5)" and looks at "Sysglobals-10(A5)" to find the location of the RECOVER-record. This location is loaded into the Stack Pointer register (SP). The location of the RECOVER routine is then popped off the stack and control is transferred to the RECOVER routine. The next value popped off the stack is the stack frame pointer for the RECOVER routine. It is moved to A6. Then the higher level RECOVER-record pointer is popped off the stack and moved to "Sysglobals-10(A5)".

Once these values have been restored, you may examine the value at "Sysglobals-2(A5)" and determine what action to take. If you want to handle the error, you may do so. If not, execute a "TRAP #10" and the problem will ripple out to be handled by the higher level RECOVER routine.

Here is the assembly version of the RECOVER routine:

```
204    4EFA 0024         jmp Rbase+242
208    2C5F              movea.l (sp)+,a6
210    2B5F FFF6         move.l (sp)+,SYSGLOBALS-10(a5)
214    7064              moveq #100,d0
216    B06D FFFE         cmp.w SYSGLOBALS-2(a5),d0
220    6600 0012         bne Rbase+240
224    4CBA 0F00         movem.w Rbase,a0-a3
       FF1C
230    48AD 0F00         movem.w a0-a3,Gbase-8(a5)
       FFF8
236    6000 0004         bra Rbase+242
240    4E4A              trap #10
242    4E5E              unlk a6
244    205F              movea.l (sp)+,a0
246    DEFC 000C         adda.w #12,sp
```

Exception Coding

In your TRY block you may wish to raise certain exception conditions and handle them in the RECOVER section. This corresponds to the Pascal standard procedure ESCAPE. When the condition is determined, store a 16-bit integer value representing the error in "SYSGLOBALS – 2(A5)" and execute a TRAP #10. For example:

```
32    4E41 FFFC        trap #1,#-4
36    202E 000C        move.l 12(a6),d0
40    D0AE 0008        add.l 8(a6),d0
44    4E76             trapv
46    2D40 FFFC        move.l d0,-4(a6)
50    4AAE FFFC        tst.l -4(a6)
54    6C00 000A        bge Rbase+66
58    3B7C 0064        move.w #100,SYSGLOBALS-2(a5)
      FFFE
64    4E4A             trap #10
66    2D6E FFFC        move.l -4(a6),16(a6)
      0010
72    4E5E             unlk a6
74    205F             movea.l (sp)+,a0
76    504F             addq.w #8,sp
```

The example generates an escape with escapecode 100 if lines 58-64 get executed. In your recovery section, check "SYSGLOBALS – 2(A5)" to see if you recognize the value. If you do, make the appropriate recovery. Otherwise, your RECOVER section restores the old RECOVER-record location and issues another TRAP #10. Thus the error is passed on to the next RECOVER block.

Returning to Pascal

When returning to Pascal from assembly, the stack must be cleaned up, a function value must be left on the top of the stack if appropriate, and all Pascal dedicated registers must be restored (A5, A6 and A7).

You can return to Pascal by leaving the return address on the top of the stack and executing an RTS, or you can store the return address in an address register and execute a JMP indirect through the address register.

```
216   B06D FFFE        cmp.w SYSGLOBALS-2(a5),d0
220   6600 0012        bne Rbase+240
224   4CBA 0F00        movem.w Rbase,a0-a3
      FF1C
230   48AD 0F00        movem.w a0-a3,Gbase-8(a5)
      FFF8
236   6000 0004        bra Rbase+242
240   4E4A             trap #10
242   4E5E             unlk a6
244   205F             movea.l (sp)+,a0
246   DEFC 000C        adda.w #12,sp
250   4ED0             jmp (a0)
252   4E75             dc.w 20085     or dc.b 78,117      or dc.b 'Nu'
```

Declaring External Procedures

Most of the subjects that have been covered in this section are relevant to EXTERNAL procedures.

If you just want to write a routine, put it in "LIBRARY" and call it from Pascal by declaring it as EXTERNAL, you won't need to be concerned with IMPORT text.

You will need to generate EXT and DEF tables. And you will have to deal with parameters. You may or may not want to deal with local variable space. If you want local space, you will reference your parameters relative to (A6). Otherwise, reference them relative to (SP). You will **not** have to write a module initialization body.

The TRY-RECOVER mechanism is also optional. There's always a RECOVER routine somewhere that has to handle those errors. The Operating System puts one around your program before execution.

You must be concerned with the stack. All the parameters must be removed. It must be left in the condition it was in before the calling procedure started preparing for the call.

You must be concerned with restoring A5 and A6 to their original values.

Write the routine, assemble it, and use the Librarian to put it in "LIBRARY". From Pascal, declare it as EXTERNAL. Call it just as if it were a Pascal procedure.

Just remember — if you're not using standard types, every program that calls this routine will have to define the special types just as you had originally defined them.

Instruction Format

In General

Assembly instructions are written one per line. Upper and lower case characters may be used interchangeably except inside of quoted strings. Instructions are free format with respect to spaces.

If a label is present, it must start in column 1 of the line. The opcode must start in column 2 or later. Blanks are not permitted within the operand field. The first blank encountered after the start of the operand field begins the comment field.

```
1234567890123456789 0 12345678901234567890

Label  MOVE  A1,A2        Comment field
```

A "*" in column 1 indicates a comment.

```
1234567890123456789 0 12345678901234567890

*
*  These are comments.
*
```

Symbols

Symbols must begin with an alphabetic character, but may contain letters, numbers, "@", "$" and "_". Symbols may contain any number of characters. The restriction is that each instruction must be contained on one line.

"*" is a symbol having the value of the location counter (except when in column 1).

Symbols are either absolute, relative, or predefined register symbols. An absolute symbol is defined as one which either follows an ORG pseudo-op or is equated to an absolute expression. A relative symbol is one which follows a RORG pseudo-op or is equated to a relative expression. The pseudo-ops REFA (absolute) and REFR (relative) are used to define the type of external symbols. Register symbols are A0...A7 and D0...D7.

Opcodes

Mnemonic operation codes (opcodes) and their syntax are defined in the *MC68000 User's Manual*. The Assembler does not allow abbreviated opcodes as does Motorola's assembler. Size suffixes are only allowed for those operations which include a size field in the instruction and for the Conditional Branch (Bcc).

Size Suffixes

Size suffixes are used in the language to specify the size of the operand in the instruction, including addressable locations and registers. All instructions that can operate on more than one data size will assume the default size of word (16-bits) unless a size suffix is used. Size suffixes can also be appended to address register specifications when used in indexed addressing.

Operand sizes are defined as follows:

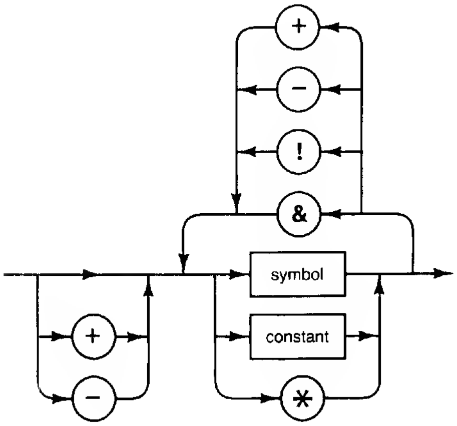
B - byte = 8 bits
W - word = 16 bits
L - long word = 32 bits

The suffix is appended to the opcode or address register specification, and separated from it by a period.

The Branch instructions (BCC, BRA, BSR) are automatically two word instructions on forward branches unless a ".S" suffix is attached to the opcode. The Assembler chooses the appropriate instruction size for backward branches.

Expressions

Expressions are limited to the operators "+", "−", "!" (bitwise OR), and "&" (bitwise AND). Expressions are evaluated in strict left to right order, and parentheses are not allowed. Only one external symbol, or symbol equated to an expression containing an external, is allowed in an expression.



Addressing Modes

Addressing modes are described in detail in the *MC68000 User's Manual.* The following descriptions are of the syntax requirements of the various addressing modes.

Register Direct

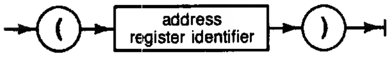
Specifies that the operand is in one of the 16 general purpose registers.

register identifier

Item	Description	Range Restrictions
register identifier	A two character mnemonic representing a processor register	A0...A7, D0...D7, SP, SR, PC

Address Register Indirect

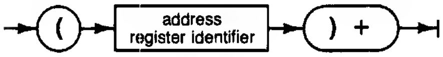
The address of the operand is in the specified address register.



Item	Description	Range Restrictions
address register identifier	A two character mnemonic representing a processor address register	A0...A7

Address Register Indirect with Postincrement

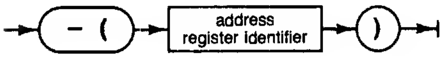
The address of the operand is in the specified address register. The contents of the address register are incremented by 1, 2, or 4 depending upon the size suffix after the operand is used.



Item	Description	Range Restrictions
address register identifier	A two character mnemonic representing a processor address register	A0...A7

Address Register Indirect with Predecrement

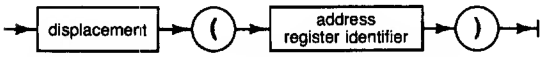
The address of the operand is in the specified address register. The contents of that register are decremented by 1,2, or 4 depending upon the size suffix before the operand is used.



Item	Description	Range Restrictions
address register identifier	A two character mnemonic representing a processor address register	A0...A7

Address Register Indirect with Displacement

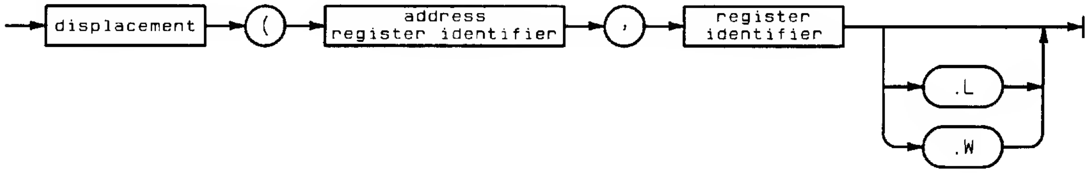
The address of the operand is the sum of the address in the specified address register and the 16-bit sign extended displacement.



Item	Description	Range Restrictions
displacement	Expression	-2^{15} thru $2^{15}-1$
address register identifier	A two character mnemonic representing a processor address register	A0...A7

Address Register Indirect with Index

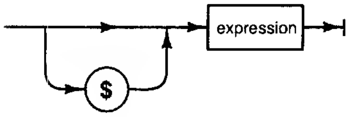
The address of the operand is the sum of the address in the specified address register and the 8-bit sign extended displacement and the contents of the specified index (A or D) register. The index is used as a 16-bit sign extended value unless ".L" is appended to the register specification. A displacement must be specified.



Item	Description	Range Restrictions
displacement	Expression	-2^7 thru 2^7-1
address register identifier	A two character mnemonic representing a processor address register	A0...A7
register identifier	A two character mnemonic representing a processor register	A0...A7, D0...D7, SP,

Absolute Short Address

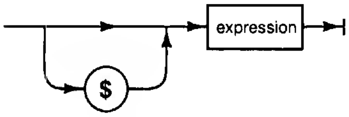
A 16-bit sign extended address.



Item1	Description	Range Restrictions
expression	Absolute expression	-2^{15} thru 2^{15}-1

Absolute Long Address

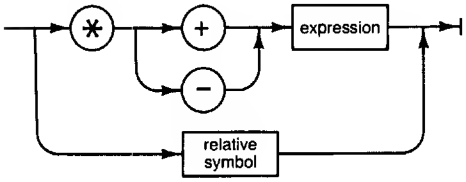
A 32-bit address



Item	Description	Range Restrictions
expression	Absolute expression	-2^{31} thru 2^{31}-1

Program Counter with Displacement

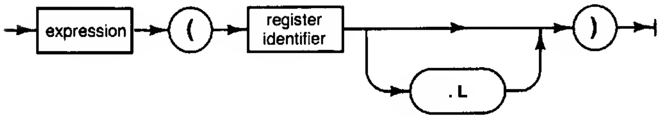
The address of the operand is the sum of the program counter (*) and the sign extended 8 or 16-bit displacement integer.



Item	Description	Range Restrictions
expression	Absolute expression	-2^{15} thru $2^{15}-1$
relative symbol	Symbol within the program	See "Symbols"

Program Counter with Index

The address of the operand is the sum of the program counter, the sign extended 8-bit displacement value, and the value in the specified index (A or D) register. Only the low 16 bits of the index register will be used unless .L is appended to the index register specification.



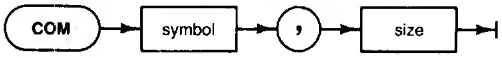
Item	Description	Range Restrictions
expression	Relocatable expression	-2^{31} thru $2^{31}-1$ expression must evaluate to within ± 128 of the current PC value

Assembler Pseudo-Op Reference

The following is a list of the commands which direct the assembler to take the described actions. For a list of the Assembler-language and machine-language instructions, see the *MC68000 User's Manual.*

COM

Used to define a global area.



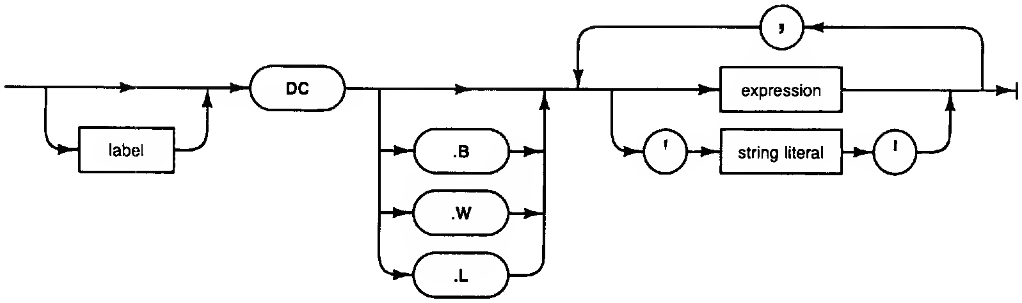
Item	Description	Range Restrictions
symbol	An identifier for the global area	see "Symbols"
size	A numeric expression	-2^{31} thru $2^{31}-1$

Semantics

The exact location of the global area will be determined at link time. The symbol is DEFined as an entry point. The amount of space is specified by the absolute value of the expression. If size is negative, the value of the symbol will be the offset from (A5) to the top of the global area and variables will have negative offsets from the symbol. This is how the Compiler does it. If size is positive, the symbol's value will be the bottom of the area, relative to (A5), and offsets will be positive. Only one COM statement allowed per assembly.

DC

Used to define some constant value or values, including string literals, and place them in storage.



Item	Description	Range Restrictions
label	An identifier for the constant	see "Symbols"
value	An expression that can be evaluated in pass 1	-2^{31} thru $2^{31}-1$
string literal	A string of characters	The instruction must be contained on one line

Semantics

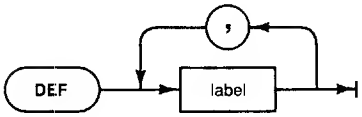
Size suffixes may be used to specify the units of storage into which the values will be justified for storage. In the case of string literals, the amount of storage needed will be determined by the Assembler and each character will be assigned into a unit.

DECIMAL

Causes addresses in the listing to be printed inn decimal rather than in hex notation.

DEF

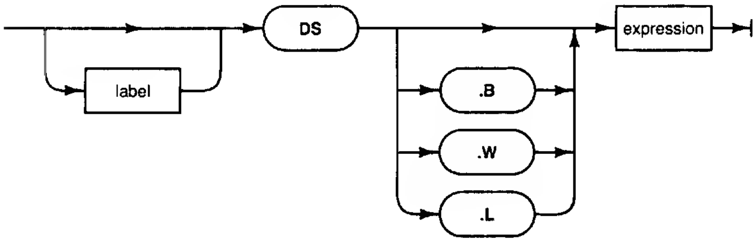
Defines a label or list of labels as entry points for other modules.



Item	Description	Range Restrictions
label	An entry point identifier	see "Symbols"

DS

Reserves storage space.



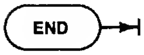
Item	Description	Range Restrictions
label	An identifier for the data space	see "Symbols"
number	An expression that can be evaluated in pass 1	0 thru $2^{31}-1$

Semantics

The units of space are specified by the size suffix. The number of units is determined by the expression.

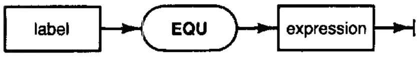
END

Indicates the end of the assembly. This should be the last line of the assembly.



EQU

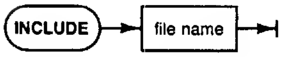
Assigns the value and attribute (absolute or relative) of the expression to the label.



Item	Description	Range Restrictions
label	An identifier	see "Symbols"
value	An expression that can be evaluated in pass 1	-2^{31} thru $2^{31}-1$

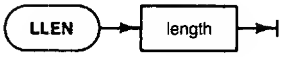
INCLUDE

Specifies a file to be merged into the assembly at the point where the instruction is located. The '.TEXT' suffix will be automatically appended to the file name. The INCLUDEd file may not contain another INCLUDE.



LLEN

Used to specify the line length (column width) of your printer.

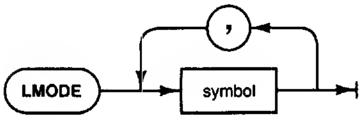


LIST

Turns the printer listing back on. You must have requested a listing when the Assembler was initiated for this to have an effect. LIST is used with NOLIST to exclude blocks of text from the listing.

LMODE

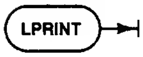
Specifies a symbol or list of symbols to be accessed using long absolute addressing mode. Overrides short addressing and PC-relative mode implications of REFR, ORG, and RORG.



Item	Description	Range Restrictions
symbol	A location identifier	see "Symbols"

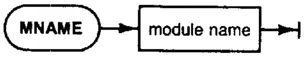
LPRINT

(Default) Causes all output from DC statements to be printed. (See SPRINT)



MNAME

Used to assign a name to an Assembler module. The default is to assign the file name to the module.



NOLIST

Turns off the listing until a LIST is encountered.

NOOBJ

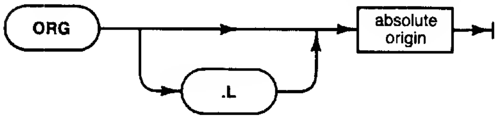
Requests that no object code be produced.

NOSYMS

Inhibits the listing of the symbol table at the end of the program.

ORG

Specifies an absolute origin. When used with the ".L" option, it forces long mode addressing for forward and external references. Otherwise short absolute addressing mode is implied.



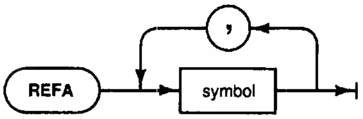
Item	Description	Range Restrictions
absolute origin	A numeric expression that can be evaluated in pass 1	-2^{31} thru $2^{31}-1$

PAGE

Advances listing to top of next page. This command will not be printed on the listing.

REFA

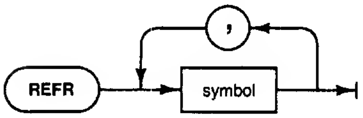
Defines a symbol or list of symbols as external and absolute references. The size of the effective address is implied by the ORG statement.



Item	Description	Range Restrictions
symbol	A location identifier	see "Symbols"

REFR

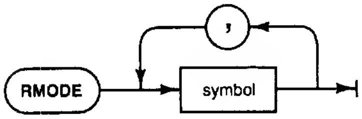
Defines a symbol or list of symbols as external and PC relative references.



Item	Description	Range Restrictions
symbol	A location identifier	see "Symbols"

RMODE

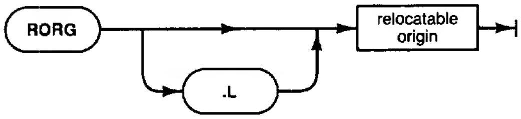
Specifies a symbol or list of symbols for access using PC-relative addressing. Overrides all other addressing mode specifications.



Item	Description	Range Restrictions
symbol	A location identifier	see "Symbols"

RORG

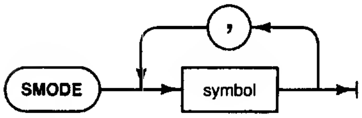
Sets a relocatable origin. Using the 'L' option, forces long absolute addressing mode for forward and external references. Otherwise, PC-relative addressing mode is implied for forward references and short absolute addressing mode for REFA symbols.



Item	Description	Range Restrictions
relocatable origin	a numeric expression that can be evaluated in pass 1	-2^{31} thru $2^{31}-1$

SMODE

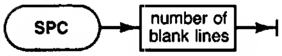
Specifies a symbol or list of symbols to be accessed using short absolute addressing mode. Overrides all other addressing mode specifications.



Item	Description	Range Restrictions
symbol	A location identifier	see "Symbols"

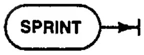
SPC

Directs the assembler to generate the specified number of blank lines. Used to separate blocks of code or blocks of comments on the listing.



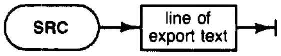
SPRINT

Print only the first line of output for the DC statements. Otherwise, each word used to store the constant is printed.



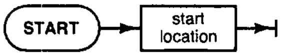
SRC

Used to specify the IMPORT text information which the Compiler needs when importing the module. Use one SRC for each line of IMPORT text. (see programming section)



START

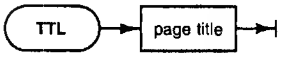
Specifies a start location for execution of the main program. Use only in the main program.



Item	Description	Range Restrictions
start location	An integer numeric or symbolic expression	-2^{31} thru $2^{31}-1$

TTL

Specifies a title to appear on each page of the assembler listing.



The Examples

Listings of the two programs and two modules are given here and also have been provided on the documentation disc (DOC:). On the disc they are provided in source and object form. The file (ASMB_P1) imports the file (ASMB_M1). These are both Pascal files. The Pascal file (ASMB_P2) imports the Assembler language file (ASMB_M2).

If you want to see them work, you must either use the Librarian to link the modules to the programs, P-load the modules, or put the modules in the current System Library. You can then execute the two programs.

The Sample Pascal Programs

This Program Imports the Pascal Module

```
$search '#3:ASMB_M1
Program test(input,output);
Import simple;
var i,j,k : rec;
begin
   initialize;
   i.i1:=1; i.i2:=2;
   j.i1:=3; j.i2:=4;
   add(i,j,k);
   writeln(k.i1,k.i2)
end.
```

This Program Imports the Assembly Module

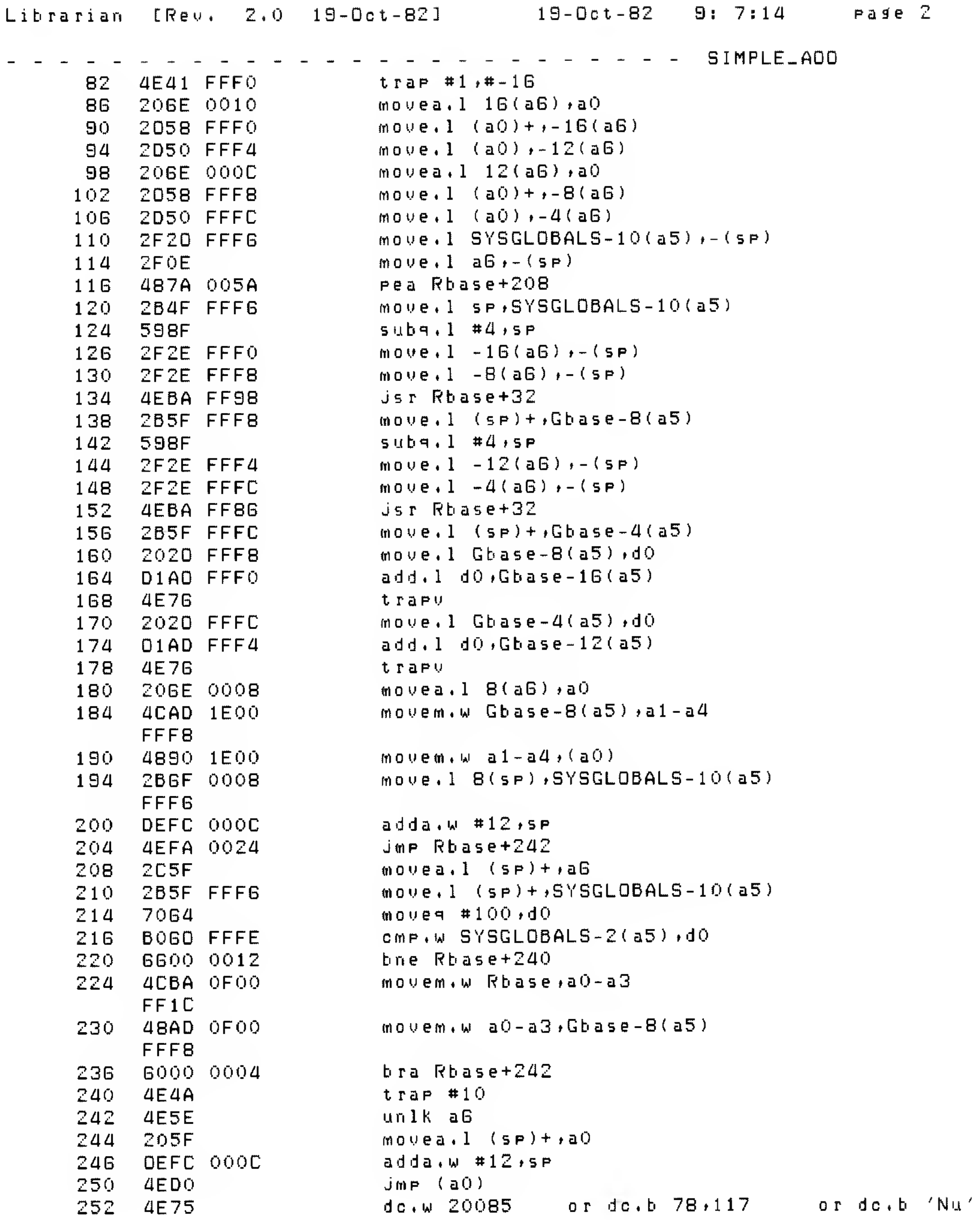
```
$search '#3:ASMB_M2
Program test(input,output);
Import simple2;
var i,j,k : rec;
begin
   initialize;
   i.i1:=1; i.i2:=2;
   j.i1:=3; j.i2:=4;
   add(i,j,k);
   writeln(k.i1,k.i2)
end.
```

The Sample Pascal Module

```
$sysprog$        (*to enable try-recover*)

module simple;
export
  type
    rec = record
             i1: integer;
             i2: integer;
          end;
  const
    zero = rec [i1:0,i2:0];
  var
    lastresult: rec;

  procedure initialize;
  procedure add (a,b: rec; var out: rec);

implement
  var
    sum: rec;

  procedure initialize;
    begin  sum := zero  end;

  function partadd (x,y: integer): integer;
  var  temp: integer;
  begin
    temp := x+y;
    if temp < 0 then escape(100);
    partadd := temp;
  end; (*partadd*)

  procedure add (a,b: rec; var out: rec);
  begin
    try
      lastresult.i1 := partadd(a.i1,b.i1);
      lastresult.i2 := partadd(a.i2,b.i2);
      sum.i1 := sum.i1+lastresult.i1;
      sum.i2 := sum.i2+lastresult.i2;
      out := lastresult;
    recover
      if escapecode = 100
        then lastresult := zero
        else escape(escapecode);
  end; (*add*)

end.
```

The Disassembly of the Module

```
Librarian  [Rev.  2.0  19-Oct-82]        19-Oct-82   9: 7:14      page 1

MODULE     SIMPLE     Created  8-Oct-82
NOTICE:  (none)
  Produced by Pascal Compiler of 20-Sep-82
  Revision number 2
  Directory size     172 bytes
  Module size       3072 bytes
  Module NOT executable
  Code base                  0      Size          254 bytes
  Global base                0      Size           16 bytes
  EXT     block    5      Size          20 bytes
  DEF     block    3      Size         114 bytes
  EXPORT block     1      Size         192 bytes
  There are              1 TEXT records


TEXT RECORD #            1  of 'SIMPLE':
  TEXT start block     2       Size         254 bytes
  REF  start block     4       Size          42 bytes
  LOAD address      Rbase

       0  0000               dc.w 0        or dc.b 0,0        or dc.b '  '
       2  0000               dc.w 0        or dc.b 0,0        or dc.b '  '
       4  0000               dc.w 0        or dc.b 0,0        or dc.b '  '
       6  0000               dc.w 0        or dc.b 0,0        or dc.b '  '
       8  0000               dc.w 0        or dc.b 0,0        or dc.b '  '
- - - - - - - - - - - - - - - - - - - - - - - - - - -  SIMPLE_INITIALIZE
      10  4E41 0000          trap #1,#0
      14  4CBA 0F00          movem.w Rbase,a0-a3
          FFEE
      20  48AD 0F00          movem.w a0-a3,Gbase-16(a5)
          FFF0
      26  4E5E               unlk a6
      28  4E75               rts
      30  0000               dc.w 0        or dc.b 0,0        or dc.b '  '

- - - - - - - - - - - - - - - - - - - - - - - - - - -
      32  4E41 FFFC          trap #1,#-4
      36  202E 000C          move.l 12(a6),d0
      40  D0AE 0008          add.l 8(a6),d0
      44  4E76               trapv
      46  2D40 FFFC          move.l d0,-4(a6)
      50  4AAE FFFC          tst.l -4(a6)
      54  6C00 000A          bge Rbase+66
      58  3B7C 0064          move.w #100,SYSGLOBALS-2(a5)
          FFFE
      64  4E4A               trap #10
      66  2D6E FFFC          move.l -4(a6),16(a6)
          0010
      72  4E5E               unlk a6
      74  205F               movea.l (sp)+,a0
      76  504F               addq.w #8,sp
      78  4ED0               jmp (a0)
      80  0000               dc.w 0        or dc.b 0,0        or dc.b '  '
```



```
Librarian  [Rev.  2.0  19-Oct-82]       19-Oct-82   9: 7:14      page 2

- - - - - - - - - - - - - - - - - - - - - - - - - - - - SIMPLE_ADD
      82  4E41 FFF0          trap #1,#-16
      86  206E 0010          movea.l 16(a6),a0
      90  2058 FFF0          move.l (a0)+,-16(a6)
      94  2D50 FFF4          move.l (a0),-12(a6)
      98  206E 000C          movea.l 12(a6),a0
     102  2058 FFF8          move.l (a0)+,-8(a6)
     106  2D50 FFFC          move.l (a0),-4(a6)
     110  2F2D FFF6          move.l SYSGLOBALS-10(a5),-(sp)
     114  2F0E               move.l a6,-(sp)
     116  487A 005A          pea Rbase+208
     120  2B4F FFF6          move.l sp,SYSGLOBALS-10(a5)
     124  598F               subq.l #4,sp
     126  2F2E FFF0          move.l -16(a6),-(sp)
     130  2F2E FFF8          move.l -8(a6),-(sp)
     134  4EBA FF98          jsr Rbase+32
     138  2B5F FFF8          move.l (sp)+,Gbase-8(a5)
     142  598F               subq.l #4,sp
     144  2F2E FFF4          move.l -12(a6),-(sp)
     148  2F2E FFFC          move.l -4(a6),-(sp)
     152  4EBA FF86          jsr Rbase+32
     156  2B5F FFFC          move.l (sp)+,Gbase-4(a5)
     160  202D FFF8          move.l Gbase-8(a5),d0
     164  D1AD FFF0          add.l d0,Gbase-16(a5)
     168  4E76               trapv
     170  202D FFFC          move.l Gbase-4(a5),d0
     174  D1AD FFF4          add.l d0,Gbase-12(a5)
     178  4E76               trapv
     180  206E 0008          movea.l 8(a6),a0
     184  4CAD 1E00          movem.w Gbase-8(a5),a1-a4
          FFF8
     190  4890 1E00          movem.w a1-a4,(a0)
     194  2B6F 0008          move.l 8(sp),SYSGLOBALS-10(a5)
          FFF6
     200  DEFC 000C          adda.w #12,sp
     204  4EFA 0024          jmp Rbase+242
     208  2C5F               movea.l (sp)+,a6
     210  2B5F FFF6          move.l (sp)+,SYSGLOBALS-10(a5)
     214  7064               moveq #100,d0
     216  B06D FFFE          cmp.w SYSGLOBALS-2(a5),d0
     220  6600 0012          bne Rbase+240
     224  4CBA 0F00          movem.w Rbase,a0-a3
          FF1C
     230  48AD 0F00          movem.w a0-a3,Gbase-8(a5)
          FFF8
     236  6000 0004          bra Rbase+242
     240  4E4A               trap #10
     242  4E5E               unlk a6
     244  205F               movea.l (sp)+,a0
     246  DEFC 000C          adda.w #12,sp
     250  4ED0               jmp (a0)
     252  4E75               dc.w 20085    or dc.b 78,117     or dc.b 'Nu'
```

The Assembly Language Module

```
          mname simple2

          src module simple2;
          src export
          src   type
          src      rec = record
          src         i1 : integer;
          src         i2 : integer;
          src      end;
          src   const
          src      zero = rec[i1:0,i2:0];
          src   var
          src      lastresult : rec;
          src   procedure initialize;
          src   procedure add(a,b : rec;
                          var out : rec);
          src end;

          com simple2,-16

          def simple2_add
          def simple2_initialize
          def simple2_zero,simple2_simple2

          refa sysglobals

lastresult       equ simple2-8
lastresult_i1    equ simple2-8
lastresult_i2    equ simple2-4
sum              equ simple2-16  (all are relative to a5)
sum_i1           equ simple2-16
sum_i2           equ simple2-12
escapecode       equ sysglobals-2
recover_rec      equ sysglobals-10

          rorg 0

simple2_zero     dc.l   0,0


simple2_initialize  trap #1      (stack check)
          dc.w   0      (no local space)

          movem.l simple2_zero,a0-a1

          movem.l a0-a1,sum(a5)


          unlk a6
          rts

simple2_partadd  trap #1
          dc.w   -4

result           equ 16
x                equ 12
y                equ 8      (all are relative to a6)
ret_addr         equ 4
```

```
dyn_link        equ 0
temp            equ -4

        move.l x(a6),d0  (temp:=x+y)
        add.l  y(a6),d0
        trapv    (overflow check)
        move.l d0,temp(a6)

        tst.l  temp(a6)  (if temp<0)
        bge past_escape
        move #100,escapecode(a5)

        trap #10     (then escape 100)

past_escape     move.l temp(a6),result(a6)

*                             (partadd:=temp)

        movea.l ret_addr(a6),a0
        unlk a6
        adda.l  #12,sp

        jmp (a0)

simple2_add      trap #1     (stack check)
        dc.w -16   (for param copies)

a_addr          equ 16
b_addr          equ 12
out_addr        equ 8
ret_addr2       equ 4
dyn_link2       equ 0      (relative to a6)
b_i2_copy       equ -4
b_i1_copy       equ -8
a_i2_copy       equ -12
a_i1_copy       equ -16

        movea.l a_addr(a6),a0    (making local
        move.l (a0)+,a_i1_copy(a6)   copies)
        move.l (a0),a_i2_copy(a6)
        movea.l b_addr(a6),a0
        move.l (a0)+,b_i1_copy(a6)
        move.l (a0),b_i2_copy(a6)

        move.l recover_rec(a5),-(sp)   (TRY)
        move.l a6,-(sp)
        pea    recover_addr
        move.l sp,recover_rec(a5)

        subq.l #4,sp   (calling partadd)
        move.l a_i1_copy(a6),-(sp)
        move.l b_i1_copy(a6),-(sp)
        jsr  simple2_partadd
        move.l (sp)+,lastresult_i1(a5)

        subq.l #4,sp   (calling partadd)
        move.l a_i2_copy(a6),-(sp)
```

```
         move.l b_i2_copy(a6),-(sp)
         jsr  simple2_partadd
         move.l (sp)+,lastresult_i2(a5)

         move.l lastresult_i1(a5),d0   (sum:=
         add.l  d0,sum_i1(a5)        sum+lastresult)
         trapv
         move.l lastresult_i2(a5),d0
         add.l  d0,sum_i2(a5)
         trapv

         movea.l  out_addr(a6),a0
         movem.l  lastresult(a5),a1-a2

         movem.l  a1-a2,(a0)      (out:=lastresult)

         move.l 8(sp),recover_rec(a5)

         adda.l #12,sp          (end of TRY)

         jmp  past_recover

recover_addr    movea.l  (sp)+,a6   (RECOVER)
         move.l  (sp)+,recover_rec(a5)
         moveq  #100,d0   (if escapecode=100)
         cmp.w  escapecode(a5),d0
         bne   sys_error
         movem.l  simple2_zero,a0-a1

*                              (then lastresult:=0)
         movem.l  a0-a1,lastresult(a5)

         bra  past_recover
sys_error        trap #10        (else escape)

past_recover    unlk  a6
         movea.l (sp)+,a0
         adda.l  #12,sp

         jmp (a0)

simple2_simple2   rts    (initialization body)

         end
```

```
 *** 68000 ASSEMBLER SYMBOL TABLE DUMP ***

            EXTERNAL SYMBOLS

    SYMBOL       TYPE    DEF       VALUE
SIMPLE2          ABS       19     00000001
SYSGLOBALS       ABS       25     00000002

            INTERNAL SYMBOLS

    SYMBOL       TYPE    DEF       EQU SYM           VALUE
A0               AREG      0                        00000000
A1               AREG      0                        00000001
A2               AREG      0                        00000002
A3               AREG      0                        00000003
A4               AREG      0                        00000004
A5               AREG      0                        00000005
A6               AREG      0                        00000006
A7               AREG      0                        00000007
A_ADDR           ABS      80                        00000010
A_I1_COPY        ABS      88                        FFFFFFF0
A_I2_COPY        ABS      87                        FFFFFFF4
B_ADDR           ABS      81                        0000000C
```

```
B_I1_COPY       ABS        86                        FFFFFFF8
B_I2_COPY       ABS        85                        FFFFFFFC
CCR             STREG       0                        00000005
D0              DREG        0                        00000000
D1              DREG        0                        00000001
D2              DREG        0                        00000002
D3              DREG        0                        00000003
D4              DREG        0                        00000004
D5              DREG        0                        00000005
D6              DREG        0                        00000006
D7              DREG        0                        00000007
DYN_LINK        ABS        56                        00000000
DYN_LINK2       ABS        84                        00000000
ESCAPECODE      ABS        33 SYSGLOBALS        +    FFFFFFFE
LASTRESULT      ABS        27 SIMPLE2           +    FFFFFFF8
LASTRESULT_I1   ABS        28 SIMPLE2           +    FFFFFFF8
LASTRESULT_I2   ABS        29 SIMPLE2           +    FFFFFFFC
OUT_ADDR        ABS        82                        00000008
PAST_ESCAPE     REL        69                        0000003E
PAST_RECOVER    REL       140                        000000F4
RECOVER_ADDR    REL       129                        000000D2
RECOVER_REC     ABS        34 SYSGLOBALS        +    FFFFFFF6
RESULT          ABS        52                        00000010
RET_ADDR        ABS        55                        00000004
RET_ADDR2       ABS        83                        00000004
SIMPLE2_ADD     REL        77                        00000052
SIMPLE2_INITIALIZE REL        40                        00000008
SIMPLE2_PARTADD REL        49                        0000001C
SIMPLE2_SIMPLE2 REL       145                        00000100
SIMPLE2_ZERO    REL        38                        00000000
SP              AREG        0                        00000007
SR              STREG       0                        00000006
SUM             ABS        30 SIMPLE2           +    FFFFFFF0
SUM_I1          ABS        31 SIMPLE2           +    FFFFFFF0
SUM_I2          ABS        32 SIMPLE2           +    FFFFFFF4
SYS_ERROR       REL       138                        000000F2
TEMP            ABS        57                        FFFFFFFC
USP             STREG       0                        00000007
X               ABS        53                        0000000C
Y               ABS        54                        00000008
```

The Librarian

Chapter 7

Introduction

It may seem obvious that the Librarian's purpose is to manage libraries. However, all the things that it can do to fulfill this responsibility may not be as obvious. This chapter will help to put all of the Librarian's capabilities into perspective. The chapter first describes libraries and object modules, providing some relevant background information that will help you to understand the Librarian operations described in the latter sections of the chapter.

Here is a brief overview of the operations you can perform with the Librarian:

- Add object modules to or remove them from libraries. For instance, you can add object modules to the System Library so that the modules will be found and loaded automatically when any program that imports them is loaded for execution.
- Link the directories of the object modules in a library file. This operation reduces the file's size.
- Obtain detailed information about the object modules in a library file. For instance, you can unassemble a compiled Pascal object file and get the Assembler language object code.
- Create new system Boot files. This operation is used to create files that are found and loaded by the Boot ROM and in turn load a system.

Let's look more closely at library files, what is in them, and how to use them.

Prerequisites

This chapter presents simple examples of user modules and libraries. If you find that you want more information about modules as you read this chapter, read the sections of the Compiler and Assembler chapters that describe modules.

If you are going to be using the Librarian for purposes other than adding modules to and removing them from the System Library (usually LIBRARY) or Initialization Library (BOOT:INITLIB), then you should also be familiar with the concepts presented in the Assembler chapter.

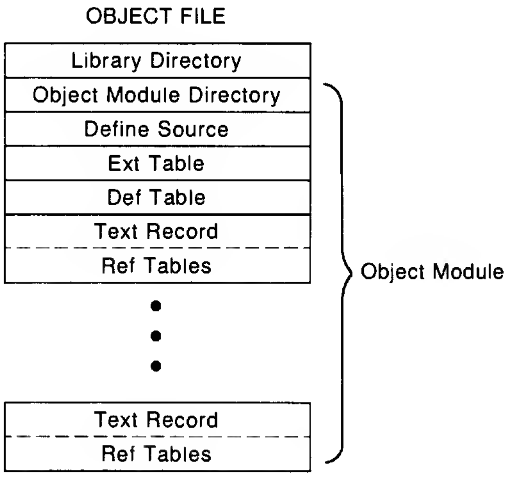
Library Overview

This section presents some important terms and concepts you will need to know in order to understand libraries. It will help you see when and why you will need to use the Librarian.

Modules and Libraries

Libraries are *object files.* They contain zero or more *object modules.* Object modules are the product of the Compiler or Assembler[1]. For instance, compiling a Pascal source module generates an object module which is placed in an object file. This file is actually a library, because it contains an object module.

An object file is composed of a *directory* of the module(s) that it contains, followed by the object modules themselves. Here is a pictorial representation of an object file.



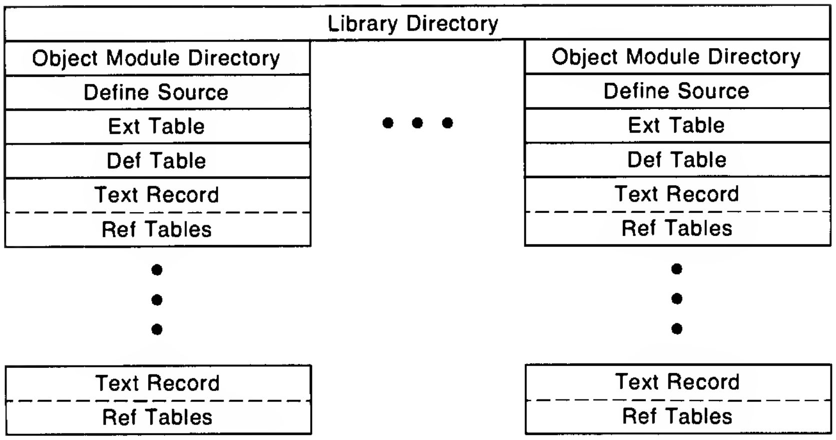
The terms Define Source, Ext Table, and so forth are defined in the Glossary of Object Code Terminology at the end of the chapter.

What the Librarian Does

The Librarian's purpose is to manage object modules. The Librarian can also produce object files; however, these files consist of object modules produced by the Compiler or Assembler. It can create library files and add modules to them or remove modules from them. The intent of these libraries is to provide a convenient location to store object modules. The following drawing shows the relationship of object modules in an object file (library):

[1] Complete descriptions of how to produce and use Pascal and Assembler modules are provided in the Compiler and Assembler chapters.

OBJECT FILE



Example Modules

For this example, we will be using three example library modules provided on the DOC: disc shipped with your system. One contains a compiled program (PROG_1.CODE), and the other two contain compiled modules (MOD_2.CODE and MOD_3.CODE).

The DOC: disc also contains the source versions of these modules. Although this chapter will only be dealing specifically with the object versions, it is a good learning experience to compile the source versions to see how the Compiler deals with imported modules. One method is briefly outlined in the next section.

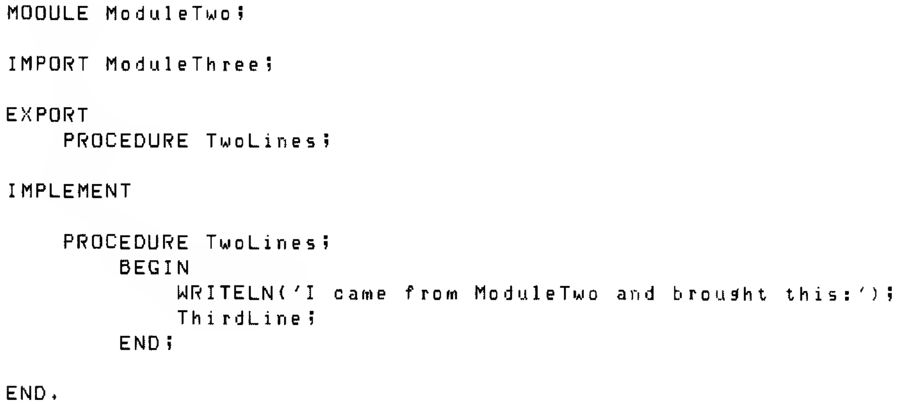
Here are source listings and brief explanations of each of the example modules.

Source Listing of PROG_1.CODE

```
PROGRAM ProgramOne(OUTPUT);

IMPORT ModuleTwo;

BEGIN
    WRITELN;
    WRITELN;
    WRITELN('*************** ProgramOne ***************');
    TwoLines;
    WRITELN('*************** ProgramOne ***************');

END.
```

The example program imports ModuleTwo, which declared the procedure named TwoLines. Here is the source of ModuleTwo, which was compiled and stored in the library (object-code) file named MOD_2.CODE.

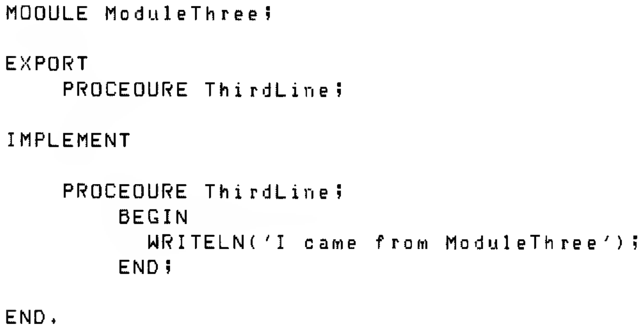
Source Listing of MOD_2.CODE



```
MOOULE ModuleTwo;

IMPORT ModuleThree;

EXPORT
    PROCEDURE TwoLines;

IMPLEMENT

    PROCEDURE TwoLines;
        BEGIN
            WRITELN('I came from ModuleTwo and brought this:');
            ThirdLine;
        END;

END.
```

ModuleTwo exports procedure TwoLines, which is used by ProgramOne. It also imports ModuleThree, which declares procedure ThirdLine and is in the library (object-code) file named MOD_3.CODE.

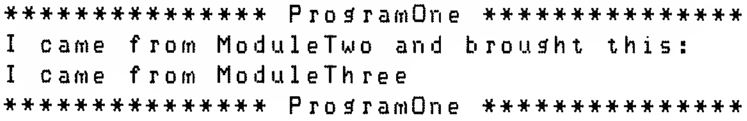
Source Listing of MOD_3.CODE



```
MOOULE ModuleThree;

EXPORT
    PROCEOURE ThirdLine;

IMPLEMENT

    PROCEOURE ThirdLine;
        BEGIN
          WRITELN('I came from ModuleThree');
        END;

END.
```

This module exports procedure ThirdLine, which is imported by ModuleTwo. Notice that it does not import any modules.

Here are the results of running the program.



```
*************** ProgramOne ***************
I came from ModuleTwo and brought this:
I came from ModuleThree
*************** ProgramOne ***************
```

Here is what happens when you run ProgramOne. First, ProgramOne prints two blank lines and then the line of asterisks that contains its name. The procedure TwoLines, imported from ModuleTwo, is then called; it prints the message: `I came from ModuleTwo and brought this:`. Procedure ThirdLine, imported from ModuleThree, is then called; it prints the message: `I came from ModuleThree`. Control is then returned to TwoLines and then to the program, which again prints out its name in asterisks.

Let's take a look at what is needed in order for you to compile and run the program.

Compiling and Running the Example Program

When a program (or module) imports modules, the imported modules must be accessible at two times:

- When the program is compiled.
- When the program is loaded and run.

Let's take a look at what happens at these two times.

How the Compiler Finds Imported Modules

At compile time, the Compiler searches for each module imported by the source program (or module); more specifically, it searches to find each module's "interface text." Here is the order of the places where the Compiler looks in search of interface text:

1. In the source text being compiled. (The source text of modules and programs can be combined into one source file, as long as the modules precede the program and are in proper sequence.)
2. In object files specified in a SEARCH Compiler option.
3. In the object file currently designated as the System Library.

(A module's interface text consists of the MODULE name, the IMPORT section, if present, and EXPORT section; these sections are part of the object module produced when the module was compiled or assembled. See the subsequent section called Getting Detailed Object File Information and the Compiler or Assembler chapters for a more complete description of interface text.)

Here is a strategy (and the method actually used) for compiling these source modules and program. (Note that you will be learning these Librarian operations in the subsequent examples given in this chapter, so you will probably want to perform this compilation exercise *after* working through the examples using the object modules and program).

1. Compile ModuleThree first (MOD_3.TEXT); call it MOD_3.CODE for simplicity. Since this module does not import any others, it will be compiled with no need to search for any imported module's interface text.
2. Use the Librarian to add the resultant object module (MOD_3.CODE) to the library file currently designated as the System Library. (Actually, you will be creating a new library into which you will place ModuleThree and the modules in the current System Library; this type of operation is subsequently explained in this chapter.)
3. After merging these two libraries (into a third new library), you will need to do one of two things: use the What command to make the resultant library the System Library; or use the Filer to change the resultant library's name back to the name of the current System Library.
4. Next, compile ModuleTwo (MOD_2.TEXT); call it MOD_2.CODE. The external references to ModuleThree will be resolved when the Compiler finds the object ModuleThree in the System Library.
5. Then place this compiled module in the System Library as in steps 2 and 3.
6. Compile the program (PROG_1.TEXT). Since both object modules upon which this program depends are in the System Library, they will be accessed automatically by the Compiler when the program is compiled.
7. Run the program. The loader automatically looks in the System Library in order to resolve the external references; it loads the modules required to complete the program (in this case, ModuleTwo and ModuleThree).

Since the program and modules have already been compiled and the object files placed on the DOC: disc, we will not discuss other alternatives of making the source files accessible to the Compiler. (However, you are again encouraged to do this after learning how to use the Librarian.)

Let's look now at how the loader finds imported object modules when the program is to be loaded for execution.

How the Loader Finds Imported Modules

Since a compiled program contains no record of where the Compiler found the imported modules, the loader must find the imported object modules at load time. Here is the order of the places where the loader looks:

1. Modules that are part of the object file being loaded.
2. In modules already P-loaded in memory, which includes all INITLIB and Operating System modules. (The loader searches for these modules in reverse order to which they were P-loaded; in other words, the most-recently loaded modules are searched first.)
3. In the current System Library file.

In order to make all imported modules part of the object file that uses them (alternative 1 above), you have two choices:

- Combine the source modules into one *source* file (and compile it). You can use the Editor to add each imported module's source file to the source program. You can also use an INCLUDE Compiler option in the source program to include each imported module's source file in the compilation of the program.
- Combine the object modules into one *object* file. Use the Librarian to combine the program and imported modules into one object file; you can optionally Link the modules to save space.

With both of these methods, only the file containing the program need be loaded; and when the program is finished, the memory used by the modules can be reclaimed for other purposes. With P-loaded modules, this is not possible (without re-booting).

If you want to P-load modules to make them accessible to the loader, you will only need to P-load all modules which are not in one of the three places stated above. In the example modules already given, ProgramOne imports ModuleTwo, and ModuleTwo imports ModuleThree. In the second example that follows, you will be creating a library that contains these two modules and then P-loading the library. (You can alternatively P-load MOD_3.CODE and MOD_2.CODE, in that order, which does not require use of the Librarian.) The loader will then be able to link the modules contained in the library to any program that imports them at execution time.

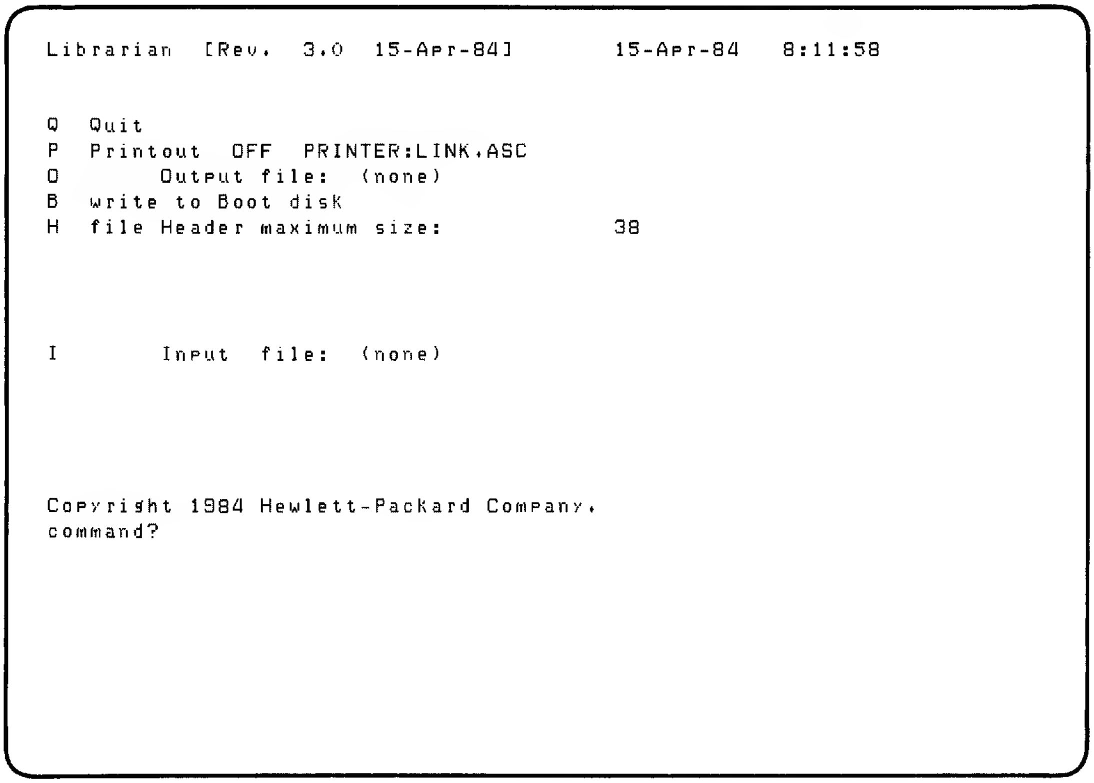
In general, the most convenient way to use modules is to place them in the file that is currently designated as the "System Library," which is the third alternative shown above. (The default System Library is the file named "LIBRARY" found on the system volume at power-up. You can also change it with the What command and the Main Command Level.) This is probably the most common reason for using the Librarian. In the first example that follows, you will add modules ModuleTwo and ModuleThree to the LIBRARY file and then run the program.

Subsequent tutorials also describe unassembling these library files and creating system Boot files.

Entering the Librarian

The Librarian is provided on the ACCESS: disc shipped with the system. To use the Librarian, you will first need to put it on-line: either place the disc labeled ACCESS: in a drive, or copy the LIBRARIAN file to another location (such as a hard disc) and use the What command (at the Main Command Level) to specify this copy as the system Librarian. After doing either of these, pressing L directs the system to load and execute the Librarian program.

Here is the Librarian's main prompt:



```
Librarian [Rev. 3.0 15-Apr-84]      15-Apr-84   8:11:58


Q  Quit
P  Printout  OFF  PRINTER:LINK.ASC
O       Output file:  (none)
B  write to Boot disk
H  file Header maximum size:        38




I       Input  file:  (none)





Copyright 1984 Hewlett-Packard Company.
command?
```

The commands shown on the left-hand side of the screen are invoked by pressing the corresponding key. You will see how to use all of them in the following tutorial discussions. All commands are summarized in the Librarian Command Reference.

Setting Up Mass Storage

You will often need two on-line mass storage volumes when using the Librarian. If you only have one volume in your system, you may need to set up a memory volume. This discusson tells why two volumes may be needed and then outlines how to estimate the size of the volumes required.

When you combine the object modules in two libraries using the Librarian, you actually create a third (new) library and then copy into it the desired modules from the other two libraries. For instance, suppose that you want to add the CONFIG:RS232 module to the BOOT:INITLIB library file. You will first create a new library, and then add the existing INITLIB modules and the RS232 module to this new library. This new library must *not* be taken off-line during the entire process.

Thus, two separate volumes are often necessary for these two reasons:

- The sum of *all* source libraries plus the new destination library often exceeds the capacity of one volume.
- The destination volume must *not* be taken off-line during this entire operation.

Continuing with the preceding example, suppose that you have only one mini-disc drive on-line (the capacity is approximately 1050 sectors). The operation cannot usually be completed, because one mini disc is not large enough to contain the modules in the INITLIB file (let's assume 750 sectors), the RS232 module (approximately 25 sectors), and the new INITLIB file (*roughly* the sum of 750 and 25 sectors). You will need two volumes for the process.

If you don't have two disc drives (or one with sufficient space), you can create a memory volume. It is usually more convenient to use the memory volume as the destination volume. In this case, you could create one with a specified size of 400 blocks, or 200 Kbytes. (Remember that memory volume blocks are *512* bytes each, while mini-disc sectors are 256 bytes each.) See the Memvol command in the Overview chapter for more specific details on creating memory volumes.

The following examples assume that either you have two disc volumes on-line or that you have created a memory volume of sufficient size. For these examples, a memory volume of 100 blocks is sufficient.

Creating Libraries of Object Modules

To create libraries, you can combine either modules provided by HP or your own modules, or any combination of the two. Let's first look at adding modules to the System Library file.

Adding Modules to the System Library

A common way to use library modules is to add them to the current System Library file. Let's assume that it is the file named LIBRARY for present purposes, although you can change it to any file by using the What command at the Main Command Level. The procedure used to add modules to LIBRARY is very similar to that of storing modules in a user library, which is the next example.

Here is a brief summary of the steps required: first, make a new library file, and copy into it all of the modules currently in LIBRARY; next, add ModuleThree and ModuleTwo to the new file (in this case the order of modules is arbitrary, since the loader will load them in the right order); then replace the LIBRARY file with this new library; execute the program, and the modules are loaded automatically for you. The actual procedure is given below.

1. Invoke the Librarian. This is done by pressing L from the Main Command Level. (If the Librarian is not on-line, insert the ACCESS: disc and try again. Remove the ACCESS: disc once the Librarian has loaded.) Now use the Librarian to create the new library.
2. Put the SYSVOL: disc (or the one containing the LIBRARY file) in the #3 drive. Press I and then type `#3:LIBRARY.` and press Return or ENTER to enter the Input file. You must include a trailing period to prevent the Librarian from appending the `.CODE` suffix.

   When the Librarian finds the Input file, the display will show the name of the first module in the file. (You should see the module named RND if you have not yet modified the LIBRARY file.) If you have a printer, you can press F to list all of the modules in the Input library.
3. (For this example, we will assume that you are using unit #4 as the second volume; however, if the LIBRARY file is small enough, you can also put the new library file on drive #3. We will also assume that the destination volume has enough room for the new library file.)

   Press O and enter `#4:NEWLIB.` as the Output file. Again, a trailing period prevents the `.CODE` suffix from being appended to the file name. If you are using a memory volume, use the unit number of the memory volume.

   (If you are using a disc, this disc must *not* be removed until you have finished creating the new NEWLIB file.)
4. Press E to enter the Edit mode. You should now see this prompt (in the middle of the screen):

   ```
   F  First module: RND
   U  Until module: (end of file)
   ```
5. You can now transfer all modules in the Input file to the Output file, including the last module, by pressing C (for Copy).
6. When the preceding transfer is complete, press A to append a module to the NEWLIB Output file. The Librarian prompts with `Input file:`. Put the DOC: disc, or whichever disc now contains ModuleThree, in Unit #3 (*not* #4, which must **not** be removed). Enter `#3:MOD_3` as the Input file.

7. The Librarian now prompts with `Enter list of modules or = for all`. Enter = for all. After ModuleThree has been transferred to the NEWLIB library, the Librarian prompts with `Append done, <space> to continue`. Press the spacebar to clear the prompt.

   Now use steps 6 and 7 again to copy ModuleTwo (in file MOD_2.CODE) into the NEWLIB file.
8. Now that all modules have been added to the NEWLIB file, press S to stop editing and K to keep the file.
9. You should now verify that the modules were indeed copied to the Output file. Press I and enter `#4:NEWLIB.` as the Input file. Press the spacebar repeatedly to scan through the modules in the new library file. If you have a printer, press F to get a File Directory listing.
10. If all modules are present, then press Q to Quit the Librarian.
11. Now you have one of two options to make this library the System Library: you can use the What command at the Main Level to specify the file named NEWLIB (on the destination volume) to be the System Library; or you can replace the LIBRARY file on the SYSVOL: disc with this file. If you choose the second option, it is probably better to keep the current copy of LIBRARY on the disc; you should first Change its name to something like OLDLIB and then Filecopy the NEWLIB file onto the SYSVOL: disc, changing its name to LIBRARY.
12. Make sure that the System Library file is on-line, and then eXecute or Run the program.

As the program is loaded, the imported modules will also be loaded automatically. Here are the results of running the program.

```
*************** ProgramOne ***************
I came from ModuleTwo and brought this:
I came from ModuleThree
*************** ProgramOne ***************
```

After the program has completed execution, the memory used by both program and modules can be used for other purposes.

As you can see, the System Library is a special library of object modules that is automatically accessed by the linking loader at program execution time (and by the Compiler at compile time). Because of this automatic access, you do not need to use the Permanent-load command to access this library's contents. This library would normally store those modules often used in your programs. Further descriptions of using HP-supplied libraries are given in the *Pascal 3.0 Procedure Library* and *Pascal 3.0 Graphics Techniques* manuals.

Making Your Own Library

Since we created a library that contains the modules named ModuleTwo and ModuleThree in the preceding example, you already know what is required to make your own library. The only difference is that you will not be adding the current LIBRARY modules to your library.

Here is a brief summary of the steps you will take in this example: first, create a new library with the Librarian and add the example modules ModuleTwo and ModuleThree to it (as with the last example, the order of modules is arbitrary; since they are in one file, the loader will take care of loading them in the proper order); P-load this library; and execute or run the program. A more detailed procedure follows.

Note

During the transfer process, you must **not** move the destination disc (the one that contains the Output file).

1. From the Main Command Level, press L to enter the Librarian. Your screen should now display the Main Prompt for the Librarian.
2. Put the destination disc in drive #4. Then press O, and type `#4:USERLIB` and press Return or ENTER to enter the Output file specification.
3. Place the DOC: Disc into the #3: disc drive. Then press I and enter `#3:MOD_3` as the Input file specification. You will see `MOD_3.CODE` displayed as the Input file. The first object module found in the object file, MODULETHREE, is also displayed. The computer is in Copying mode as shown by the word COPYING on the prompt.
5. Transfer the object module MODULETHREE using the T command. Since MODULETHREE is the only module in that file, the A command would have done the same job.
6. Repeat steps 4 and 5 to name MOD_2 as the Input file and Transfer the object module MODULETWO into your new library.
7. If you had other modules to transfer, you would repeat steps 4 and 5 as needed.
8. Press K to Keep the new file on the destination volume.
9. Press Q to Quit the Librarian and return to the Main Command Level.
10. If you P-loaded these modules as you worked through the preceding example, then you need to re-boot in order to fully test your new library (to ensure that the modules P-loaded in the preceding example aren't accessed instead).
11. Press P for the Permanent-load command. You will be prompted: `Load what code file ?` Enter `#4:USERLIB` (you don't need a period if you didn't include one when you specified this file as the Output file).
12. Now press X to eXecute ProgramOne. Answer the `Execute what file?` prompt by entering `#3:PROG_1` as the file specification.

The results of the executed program are shown below.

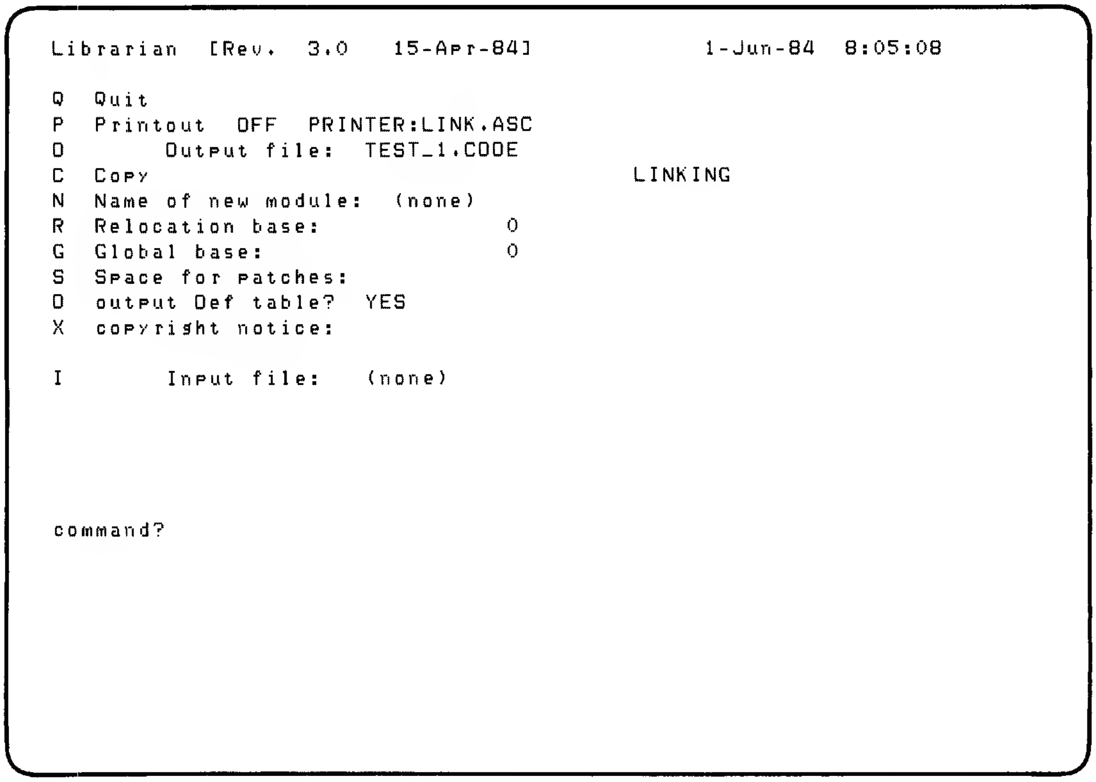
```
*************** ProgramOne ***************
I came from ModuleTwo and brought this:
I came from ModuleThree
*************** ProgramOne ***************
```

As mentioned earlier, you could also have separately P-loaded ModuleThree and ModuleTwo, in that order, and then run the program. Or, as with the preceding example, you could also have added these modules to the System Library. You could also have used What at the Main Command Level to specify this library as the System Library. The method you use depends on factors such as these: whether you are developing and testing the modules; whether you are also using other modules in the System Library; who will be using the modules; and so forth.

Linking Object Files Together

The Librarian permanently links modules together by combining their module directories into a single directory. To see this process in action, you will be linking the two example modules and the example program together.

1. Put the ACCESS: disc in a drive and press (L) to run the Librarian.
2. Put the DOC: disc in the #3 drive. Press (O), and then type `#3:TEST_1` and press (Return) or (ENTER) to enter the Output file specification. #3:TEST_1.CODE will be our linked library's name; `#3:TEST1.CODE` is now displayed as the Output file. When an Output file is named, the menu replaces the B and H command prompts with the "L Link" prompt. The Librarian also enters the COPYING mode.
3. Enter the LINKING mode by pressing (L). This is the first step in the two-step linking process. Your screen now displays a new command prompt, as shown below.



```
Librarian  [Rev.  3.0   15-Apr-84]          1-Jun-84  8:05:08

Q  Quit
P  Printout  OFF  PRINTER:LINK.ASC
O      Output file:  TEST_1.CODE
C  Copy                                  LINKING
N  Name of new module:  (none)
R  Relocation base:             0
G  Global base:                 0
S  Space for patches:
D  output Def table?  YES
X  copyright notice:

I      Input file:   (none)




command?
```

4. Press N and enter `NEWNAME` as the new module name. If you did not do this, the object module contained in the new object file, TEST_1.CODE, would be the module name of the first module transferred. To avoid confusion, use "NEWNAME".
5. Press I and enter `#3:PROG_1` as the Input file. (The ".CODE" suffix is automatically appended to the Input file's name.) PROG_1.CODE is now the Input file.
6. Press A to transfer all object modules contained in PROG_1.CODE into TEST_1.CODE. Since PROG_1.CODE contains only one module, T would have done the same job.
7. Repeat steps 5 and 6 to transfer MOD_2.CODE and MOD_3.CODE. When all files are transferred, final linking must be done.
8. Press L to complete the linking process. This is the second step in the linking process. Remember that **all** object modules must be on-line when you complete the linking process.
9. Press K to Keep the new file, and press Q to Quit the Librarian and return to the Main Command Level.

To see that everything works, execute your new program. From the Main Command Prompt press X, and then enter `TEST_1` as the file specification. The ".CODE" is automatically added to the file name. Your screen should now display the the following:

```
*************** ProgramOne ***************
I came from Module_2 and brought
I camd from Module_3
*************** ProgramOne ***************
```

The benefit gained over merely combining modules into one library is that linking modules together reduces the amount of space required to store the library.

Subtle Points about Linking

There are several subtle side effects that occur when modules are linked that should be discussed here.

- When you link object modules, the interface text is removed. Thus, linked modules **cannot** be searched by the Compiler when it is attempting to satisfy IMPORT statements; however, these modules can be used by other modules at load time by P-loading them or placing them in the System Library. (Remember that you can also keep a copy of the unlinked object module which can, of course, be imported by other modules at compile time.)
- The linking process always produces relocatable object code. This code has been relocated to the values specified by Global base and Relocatable base, but it will be relocated again when it is loaded for execution. For this reason, you don't need to specify Global base and Relocatable base – just leave them zero.

- If two or more programs are linked together into one object file, the resultant file contains code with only *one* start address (rather than the two that you began with). Contrast this to the situation in which you put two programs in an object file; when this file is executed, the two programs get executed separately in the order encountered in the file. This is the reason that you cannot link the INITLIB modules together; it is actually a set of programs and modules in a library file.
- After linking, most programs will still have unsatisfied external references (such as calls to the File System read and write routines). These unsatisfied references do not cause error messages; they are satisfied by the linking loader as the program is prepared for execution. These system routines are not part of the compiled or linked program; rather, the entire operating system looks to the linking loader like a group of P-loaded user libraries.

Summary of Linking Object Files

Note

All input modules must remain on-line for the duration of steps 6 through 9. The output file must be on-line for steps 3 thru 10.

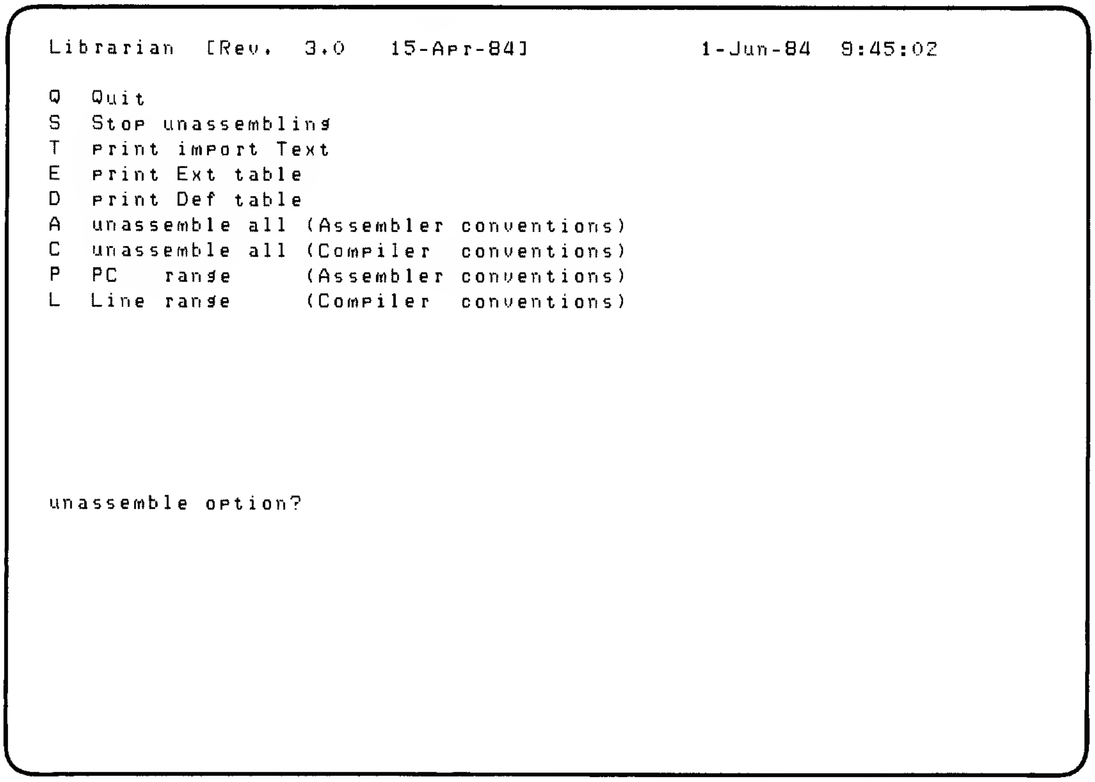
1. Enter the Librarian.
2. Be sure the disc containing the file to be linked is in the appropiate disc drive.
3. Specify an Output file name.
4. Press (L) to begin the linking process
5. Name the new module with the Name Command.
6. Specify the Input file containing the modules you want to link.
7. Transfer only those files you want into the new Output file using the All and Transfer commands.
8. Repeat steps 6 and 7 until all files are transferred.
9. Press (L) to complete the Linking process.
10. Press (K) to keep your output file.
11. Press (Q) to quit the Librarian

Getting Detailed Object File Information

Let's unassemble the file MOD_2.CODE to see how the Librarian provides detailed information about a code file. It is best to have a printer on-line while unassembling; however, if you don't, you can declare a Printout file as described in the following procedure.

1. The Librarian is on the ACCESS: disc shipped with your system. To access the Librarian, you will need to put this disc on-line, or copy the file to a disc that is on-line, or P-load the file. After the file is on-line, press L (at the Main Command Level) to load the Librarian subsystem into the computer.
2. If you don't have a printer on-line, then you must specify a file to which the unassembled information is sent. Press P (for Printout) and enter a file specification; if no suffix or trailing period is included, then ".TEXT" is appended to the file name. The screen will be updated to show that the printout device is ON and that it is the file you specified.
3. Press I and enter `#3:MOD_2` as the Input file. No Output file is needed.
4. Press U to get into the Unassemble mode.

Your screen should now show the Librarian Unassemble menu.



```
Librarian  [Rev. 3.0   15-Apr-84]            1-Jun-84  9:45:02

Q   Quit
S   Stop unassembling
T   print import Text
E   print Ext table
D   print Def table
A   unassemble all (Assembler conventions)
C   unassemble all (Compiler  conventions)
P   PC    range    (Assembler conventions)
L   Line  range    (Compiler  conventions)




unassemble option?
```

When the first command key is pressed, an information header is printed along with the desired information. This header is printed only this one time. An internal counter keeps track of the line count and prints a page heading at the top of every new page. If you change the placement of the printer paper, you may waste some paper when the counter sends a form-feed to the printer. When you quit, a final form-feed is sent to the printer automatically.

The Text and Table Commands

Use these commands to obtain Interface Text and REF and DEF tables of modules.

The Print Import (or Interface) Text Command

Pressing T prints the interface text (DEFINE SOURCE) of the module, if any. In a compiled module, the DEFINE SOURCE portion consists of the text in the MODULE, IMPORT (if present), and EXPORT declarations; in an assembled module, this text consists of the lines containing the SRC pseudo op. (Note that any comments and indentation have been removed.)

```
Librarian  [Rev.  3.0  15-Apr-84]       23-Apr-84   7: 6:51     page 1

MOOULE    MODULETWO    Created 23-Apr-84
NOTICE:  (none)
  Produced by Pascal Compiler of 23-Apr-84
  Revision number 3
  Oirectory size    174 bytes
  Module size      3072 bytes
  Module NOT executable
  Code base                 0     Size          104 bytes
  Global base               0     Size            0 bytes
  EXT     block    5      Size          72 bytes
  OEF     block    3      Size          90 bytes
  EXPORT block     1      Size          74 bytes
  There are               1 TEXT records


  OEFINE SOURCE of 'MODULETWO':

MOOULE MOOULETWO;
IMPORT ModuleThree;

EXPORT
PROCEOURE TwoLines;

END;
```

The Print EXT Table Command

Pressing E prints the table of External symbols the module references. Detailed information on the EXT table may be found later in this chapter.

```
EXT table of 'MODULETWO':

    FS_FWRITELN
    FS_FWRITEPAOC
    MODULETHREE_THIRDLINE
    SYSGLOBALS
```

The Print DEF Table Command

Pressing D prints the table of symbols the module itself defines. Detailed information on the DEF table may be found later in this chapter.

```
DEF table of 'MODULETWO':

    MODULETWO                      Gbase
    MODULETWO_MODULETWO            Rbase+102
    MODULETWO_TWOLINES             Rbase+2
    MODULETWO__BASE                Rbase
```

The Unassemble Commands

There are two conventions used when unassembling object files: Compiler and Assembler. The reason for this is that the Compiler and Assembler use different conventions for the object code that they generate.

The Compiler generates code so that each procedure begins with a `TRAP #1` or a `LINK #n,A6` and ends with a `JMP` or `RTS`. The Librarian uses this information to assume that everything from the beginning of the file to the first `TRAP #1` or `LINK` is a constant. From the end of the procedure to the next `TRAP #1` or `LINK` is also unassembled as constants. Everything else is unassembled as instructions. The Assembler convention assumes that everything is an instruction.

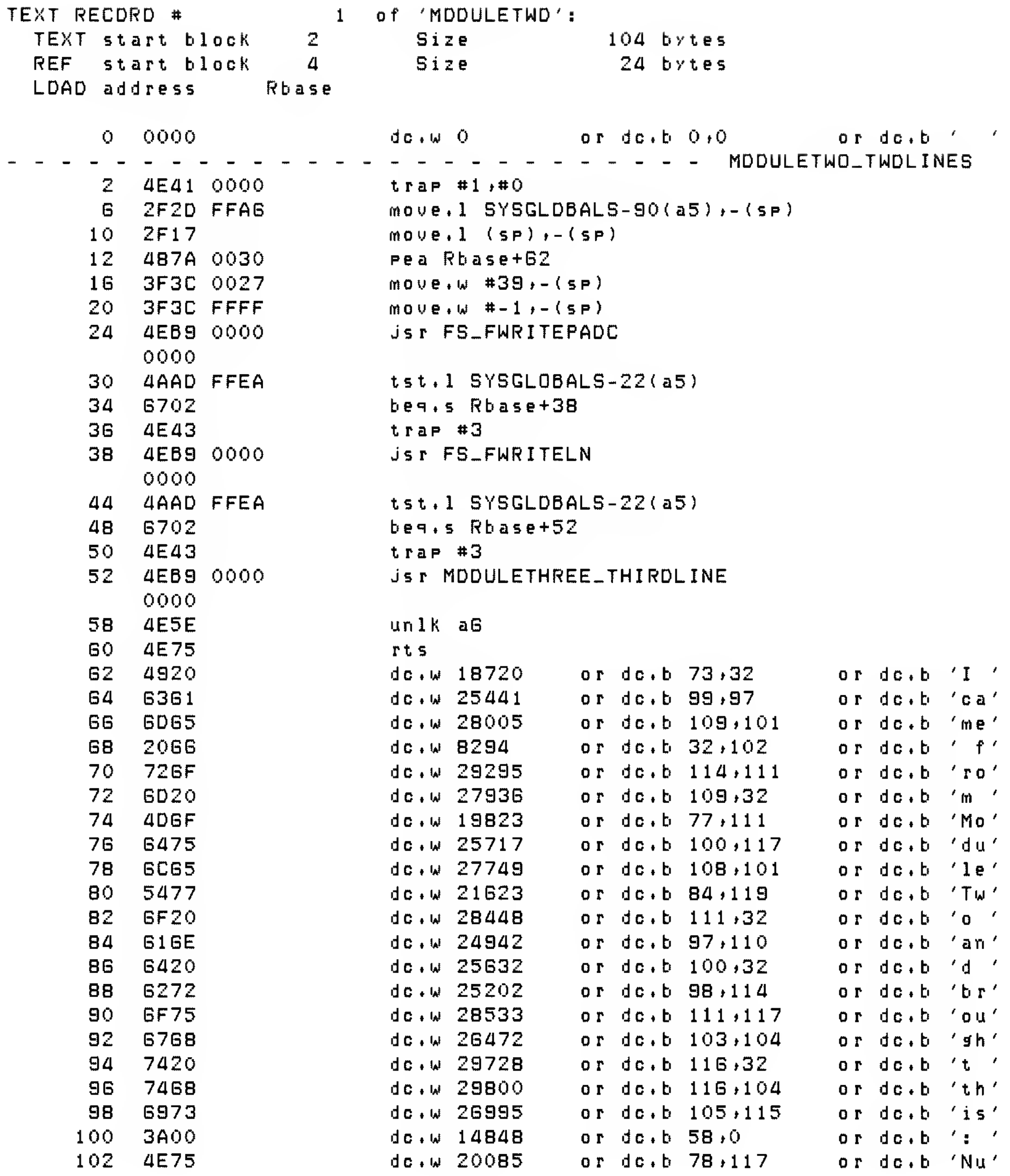
Note

All Unassemble commands require a printer unless a destination file is specified with the P command.

The Unassemble All (Compiler convention) Command

Pressing C directs the Librarian to unassemble the specified object module using the Compiler convention described above. You can use this command on files that were created by either the Assembler or Compiler. Here are the results of using this command with the MOD_2.CODE compiled object file.

```
  Librarian  [Rev.  3.0  15-Apr-84]       23-Apr-84   9:58:42      Page 1

MODULE     MODULETWO     Created 23-Apr-84
NOTICE:  (none)
  Produced by Pascal Compiler of 23-Apr-84
  Revision number 3
  Directory size    174 bytes
  Module size      3072 bytes
  Module NOT executable
  Code base                 0     Size          104 bytes
  Global base               0     Size            0 bytes
  EXT    block   5     Size          72 bytes
  DEF    block   3     Size          90 bytes
  EXPORT block   1     Size          74 bytes
  There are            1 TEXT  records
```



```
TEXT RECORD #            1  of 'MODULETWO':
  TEXT start block     2       Size           104 bytes
  REF  start block     4       Size            24 bytes
  LOAD address      Rbase

     0  0000                dc.w 0         or dc.b 0,0         or dc.b '  '
- - - - - - - - - - - - - - - - - - - - - - - - -  MODULETWO_TWOLINES
     2  4E41 0000           trap #1,#0
     6  2F2D FFA6           move.l SYSGLOBALS-90(a5),-(sp)
    10  2F17                move.l (sp),-(sp)
    12  487A 0030           pea Rbase+62
    16  3F3C 0027           move.w #39,-(sp)
    20  3F3C FFFF           move.w #-1,-(sp)
    24  4EB9 0000           jsr FS_FWRITEPAOC
        0000
    30  4AAD FFEA           tst.l SYSGLOBALS-22(a5)
    34  6702                beq.s Rbase+38
    36  4E43                trap #3
    38  4EB9 0000           jsr FS_FWRITELN
        0000
    44  4AAD FFEA           tst.l SYSGLOBALS-22(a5)
    48  6702                beq.s Rbase+52
    50  4E43                trap #3
    52  4EB9 0000           jsr MODULETHREE_THIRDLINE
        0000
    58  4E5E                unlk a6
    60  4E75                rts
    62  4920                dc.w 18720     or dc.b 73,32       or dc.b 'I '
    64  6361                dc.w 25441     or dc.b 99,97       or dc.b 'ca'
    66  6D65                dc.w 28005     or dc.b 109,101     or dc.b 'me'
    68  2066                dc.w 8294      or dc.b 32,102      or dc.b ' f'
    70  726F                dc.w 29295     or dc.b 114,111     or dc.b 'ro'
    72  6D20                dc.w 27936     or dc.b 109,32      or dc.b 'm '
    74  4D6F                dc.w 19823     or dc.b 77,111      or dc.b 'Mo'
    76  6475                dc.w 25717     or dc.b 100,117     or dc.b 'du'
    78  6C65                dc.w 27749     or dc.b 108,101     or dc.b 'le'
    80  5477                dc.w 21623     or dc.b 84,119      or dc.b 'Tw'
    82  6F20                dc.w 28448     or dc.b 111,32      or dc.b 'o '
    84  616E                dc.w 24942     or dc.b 97,110      or dc.b 'an'
    86  6420                dc.w 25632     or dc.b 100,32      or dc.b 'd '
    88  6272                dc.w 25202     or dc.b 98,114      or dc.b 'br'
    90  6F75                dc.w 28533     or dc.b 111,117     or dc.b 'ou'
    92  6768                dc.w 26472     or dc.b 103,104     or dc.b 'gh'
    94  7420                dc.w 29728     or dc.b 116,32      or dc.b 't '
    96  7468                dc.w 29800     or dc.b 116,104     or dc.b 'th'
    98  6973                dc.w 26995     or dc.b 105,115     or dc.b 'is'
   100  3A00                dc.w 14848     or dc.b 58,0        or dc.b ': '
   102  4E75                dc.w 20085     or dc.b 78,117      or dc.b 'Nu'
```

The Unassemble All (Assembler Convention) Command

Pressing (A) will cause your computer to unassemble the specified object module using the Assembler convention described above. You can use this command on files that were created by either the Assembler or Compiler.

Note

Use of the Assembler convention may produce unpredictable results, because under this convention there is no way to tell code from data. Files produced by the Compiler and unassembled under the Compiler convention will almost always produce reasonable results.

Here is the unassembly of the MOD_2.CODE object file using the Assembler convention. Notice that, with Assembler convention, the first two bytes ($0000) are assumed to be code; with Compiler convention they are assumed to be data (remember that the Compiler convention assumes that anything until the first `TRAP #1` or `LINK #n,A6` is assumed to be data). Notice also that the module heading shows that this object module was produced by the Compiler.

```
Librarian [Rev. 3.0  15-Apr-84]          23-Apr-84 10: 1:34 page 1

MODULE     MODULETWO     Created 23-Apr-84
NOTICE:  (none)
  Produced by Pascal Compiler of 23-Apr-84
  Revision number 3
  Directory size     174 bytes
  Module size       3072 bytes
  Module NOT executable
  Code base                    0      Size           104 bytes
  Global base                  0      Size             0 bytes
  EXT     block    5      Size             72 bytes
  DEF     block    3      Size             90 bytes
  EXPORT  block    1      Size             74 bytes
  There are               1 TEXT records


TEXT RECORD #             1   of 'MODULETWO':
  TEXT start block     2        Size           104 bytes
  REF  start block     4        Size            24 bytes
  LOAD address      Rbase

       0   0000 4E41          ori.b #65,d0
       4   0000 2F2D          ori.b #45,d0
       8   FFA6               dc.w -90       or dc.b 255,166     or dc.b '  '
      10   2F17               move.l (sp),-(sp)
      12   487A 0030          pea Rbase+62
      16   3F3C 0027          move.w #39,-(sp)
      20   3F3C FFFF          move.w #-1,-(sp)
      24   4EB9 0000          jsr FS_FWRITEPADC
           0000
```

```
 30  4AAD FFEA        tst.l SYSGLOBALS-22(a5)
 34  6702             beq.s Rbase+38
 36  4E43             trap #3
 38  4EB9 0000        jsr FS_FWRITELN
     0000
 44  4AAD FFEA        tst.l SYSGLOBALS-22(a5)
 48  6702             beq.s Rbase+52
 50  4E43             trap #3
 52  4EB9 0000        jsr MODULETHREE_THIRDLINE
     0000
 58  4E5E             unlk a6
 60  4E75             rts
 62  4920             lea -(a0),a4
 64  6361             bls.s Rbase+163
 66  6D65             blt.s Rbase+169
 68  2066             movea.l -(a6),a0
 70  726F             moveq #111,d1
 72  6D20             blt.s Rbase+106
 74  4D6F 6475        lea 25717(sp),a6
 78  6C65             bge.s Rbase+181
 80  5477 6F20        addq.w #2,32(sp,d6.l)
 84  616E             bsr.s Rbase+196
 86  6420             bcc.s Rbase+120
 88  6272             bhi.s Rbase+204
 90  6F75             ble.s Rbase+209
 92  6768             beq.s Rbase+198
 94  7420             moveq #32,d2
 96  7468             moveq #104,d2
 98  6973             bvs.s Rbase+215
100  3A00             move.w d0,d5
102  4E75             rts
```

The Line Range (Compiler Convention) Command

Pressing L causes two prompts to be displayed. The computer needs to know the line number range to unassemble. The code will then be unassembled up to, but not including, the upper range value. The object module must have been compiled using the $DEBUG ON$ Compiler option to be unassembled with this command.

The PC Range (Assembler Convention) Command

Pressing P causes two prompts to be displayed. The computer needs the location counter values of the segment of code you want unassembled (relative to the relocation base of the module). The code will then be unassembled up to but not including the upper location counter value.

Creating a New Boot File

At power-up, the Boot ROM searches mass storage for system Boot files: Boot ROMs 3.0 and later versions search all mass storage devices on-line and let you choose which Boot file you want; earlier Boot ROMs choose the first Boot file found on the right-hand internal disc drive. A Boot file is then loaded by the Boot ROM. The Boot file in turn may load other parts of a system. (For further details of how this system boots, see the discussion called The Booting Process in the Special Configurations chapter.)

The B command is used to create Boot files. This is an advanced option and should only be used if you have a clear understanding of system generation.

The following is an overview of the system generation process using the B command.

Note

The B command cannot create boot files in WS1.0 directories.

1. Use the Editor to produce the programs and modules that will make up the new boot program. Both Assembler language and Pascal modules may be used.
2. Assemble the Assembler language modules and compile the Pascal modules.
3. Use the Librarian to Link the code files together as desired. Be sure to specify the global and relocation bases. In addition, this file must have no unsatisfied external references. Note that the first program linked will provide the start address for the Linked file. This start address will also become the start execution address of the system Boot file at boot time.
4. Keep the linked file.
5. Specify the linked file as the Librarian Input file.
6. Now press B to properly place the module name in the destination's directory and format the code for use by the boot ROM. The B command moves the cursor up to the Output file prompt.
7. Specify the Output file as SYSTEM_xxx (the xxx can be any characters syntactically allowed for file names. With Boot ROMs 3.0 and later, the name can be SYSxxxxxxx; see Re-Naming BOOT: Files in the Special Configurations chapter for examples). Be sure to append a trailing period to the file name to keep a suffix from automatically being appended to the name.
8. Transfer the Input file into the boot file. This copies the code file.
9. Press B again to finish the Boot operation.
10. You can now either Quit the Librarian or power up and test your new system.

Librarian Command Reference

The Librarian command set consists of single-letter commands allowed when the letter prompts are displayed on the screen. You press the corresponding key to cause the command to be executed.

A In Copying and Linking modes, this command transfers All modules from the Input file to the Output file.

In Edit mode, this command is used to Append modules to the Output file.

In Unassemble mode, this command directs the Librarian to unassemble the Input file using Assembler conventions.

B The Boot command is used to create code files that are loadable by the Boot ROM. The Boot command is given instead of the Output file command. The Input modules are then combined in a format that is bootable. This command should only be used by system designers. A boot file must be a self-contained processor environment. It must be stored on a LIF or SRM volume and be named SYSTEM_xxx, where xxx represents any combination of characters. If you have Boot ROM 3.0 or later version, the file can be named SYSxxxxxxx or stored on an SRM system under the /SYSTEMS directory.

C In the Link mode, this command returns you to Copy mode. While in Copy mode, you can combine modules into a library without Linking. This mode can be used to add modules to (or remove them from) the System Library, your own library, or INITLIB (the Initialization Library file which is executed during the boot process).

In Edit mode, this command Copies the First module up to (but not including) the Until module to the Output file.

In Unassemble mode, this command unassembles the Input file according to Compiler conventions.

D In Link mode, this command controls whether or not the DEF table is included in the Output file. If the Output file is to be Linked to another file later, the DEF table must be left in the Output file. If the Output file is not to be Linked, you can save memory space by removing the DEF table. A YES includes the DEF table, a NO removes it. Pressing **D** toggles between these two choices.

In Unassemble mode, this command prints the DEF table.

E This command is available when you have specified both Input and Output files. It puts you into Edit mode, which allows you to combine modules in the Input file with Append modules and place them into the Output file (while either Copying or Linking).

In Unassemble mode, this command prints the Ext table.

F This command prints the File directory of the Input file on the current Printout file (external printer or file). It doesn't matter whether the `Printout` prompt is `ON` or `OFF`; the printout will be sent to the Printout file.

In Edit mode, this command is used to specify the First module to be transferred to the Output file. (The First module must precede the Until module in the Input file.)

G In Linking mode, designate the Global base address (most useful when preparing a file for use as a system Boot file).

H The Header command allows you to change the size of the library header. From 1 through 18 module entries require only one header block, so a header size of 18 is the minimum. If you specify less (but not 0), you will still be given 18. From 19 through 38 module entries requires two header blocks. This is the default header size. The specification is made in units of module entries – not blocks. The Librarian calculates how many blocks are necessary to maintain the number of modules you specify and then gives you the maximum number of entries that will fit in that number of blocks.

I Name the Input file containing the modules you want to transfer to the Output file. ".CODE" is automatically appended to the file name unless suppressed by a trailing period (or by the presence of another standard suffix). This prompt can be used many times to collect modules from several object files into your new object file.

K Keep the Output file. Close and lock it into the directory, purging any old file of the same name.

L Enter Linking mode or finish Linking.

In Unassemble mode, you can use this command to unassemble a section of code defined by two Line values using the Compiler convention. The code must have been compiled using the $DEBUG ON$ Compiler option.

M Enter the specific Module name you want to transfer. (The first module in the Input file is automatically displayed when an Input file is specified.)

N In Linking mode, name the New object module to be created by the Librarian. If you do not specify specify a new module name, the name of the first module transferred will be used.

O Name your Output file. ".CODE" is appended automatically unless suppressed by a trailing period (or standard suffix) in the file name. This Output file must remain on-line throughout the process of transferring modules to it.

P This command is used to turn the Printout option ON or OFF (the default is OFF) and to select a Printout file. Pressing P and Return or ENTER turns the option ON. With the option ON, the device to which the information is sent is shown on the screen. The default Printout file specification is "PRINTER:LINK.ASC" unless you specify another. ".TEXT" is automatically appended to the file name unless it is suppressed by a trailing period (or standard suffix) in the file name.

Before any information is sent to the Printout file, the Librarian first sends heading information to the device. When Linking, you will get a map of all Linking done by the Librarian. This option does not affect any information sent to the Printout file by the Unassemble commands.

In Unassemble mode, this command allows you to unassemble (using Assembler conventions) a section of code defined by two location counter range values.

Q — Quit the Librarian and return to the Main Command Level.

R — In Linking mode, designate the Relocation base address to be used.

S — In Linking mode, this command assigns Space for patches. To save execution time and memory space, the Compiler can be made to use PC-Relative addressing instead of Long-Absolute addressing. This is done with the Compiler option $CALLABS OFF$. The PC-Relative addressing mode has an address range of − 32 768 through 32 767 bytes; if the referenced procedure is out of this range, an error will occur at load or link time. This error prints an error message naming the module having the link out of range. To fix this, relink the modules adding patch space between them as needed. The number of bytes needed depends on the particular module. As a rule of thumb, begin with a patch of 100 bytes.

In Edit or Unassemble modes, this command Stops the Edit or Unassemble session and returns to the Librarian's main prompt. (This will not stop an ongoing Unassemble; however, the Stop key will.)

T — In Copying and Linking modes, this command Transfers the object module currently named in the Transfer prompt to the Output file.

In Unassemble mode, this command prints the interface Text (DEFINE SOURCE) of the current Input module.

U — Enter the Unassemble mode.

In Edit mode, this command allows you to specify the Until module. If you enter a null response (by pressing Return or ENTER with no file specification), then `<end of file>` is displayed; a subsequent Copy will copy all remaining modules in the Input file (i.e., up to the end of the file) to the Output file.

V — This command gets you into the Verify mode. This mode displays the name of each module in the Input file and allows you to Transfer it to the Output file (press T), to Unassemble it, or to not transfer or unassemble it (press the space bar) and step to the next module name. After all module names have been displayed, you automatically leave this mode. To re-verify the file's contents, press V again.

X — In Linking mode, this command allows you to enter a copyright notice as part of the Output file. The notice is part of the heading information sent to the Printout file. The notice can be up to 255 characters long.

Glossary of Object Code Terminology

Here are detailed definitions of the terms used in this manual regarding object-code library files.

DEF table (Definition Symbol Table)

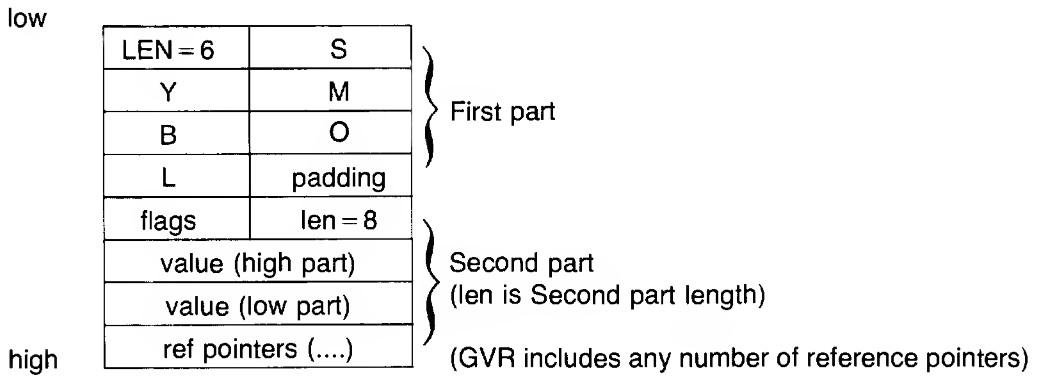
There is only one DEF table per module. It contains one DEF record for each symbol which is exported from the module. The DEF table begins on a block boundary which is specified in the directory for the module. Its length is also given in the directory. The DEF table is contiguous over its length, which means that individual DEF records within the table may cross block boundaries.

Each DEF record has two parts. The first part is a packed string containing the name of the symbol which is defined. The string begins and ends on a word (even-byte) boundary. If the string length is odd, then an extra byte is added to the end for padding so that the next part of the DEF record will begin on a word boundary.

The second part of a DEF record is a general value or address record (GVR) which defines the value of the symbol which is being exported. GVR is defined later in this section.

The value extension is 4 bytes or 8 bytes long, according to the data size field. The value of the symbol is defined to be the value extension plus what ever references are specified by the primary type and any Reference Pointers that may exist. The value extension must be present.

DEF record



LEN=6	S
Y	M
B	O
L	padding
flags	len=8
value (high part)	
value (low part)	
ref pointers (....)	

low
high
First part
Second part
(len is Second part length)
(GVR includes any number of reference pointers)

DEFINE SOURCE

This is the section of an object module that is searched by the Compiler when the module is imported (also called "interface text"). With Pascal modules, the DEFINE SOURCE consists of the declarations made by the reserved words MODULE, IMPORT (if present), and EXPORT. With Assembler modules, it consists of the lines defined by the SRC pseudo op, which are intended to serve the same function as in Pascal modules (however, it may be any arbitrary text).

There may be one table of DEFINE SOURCE per module. It begins on a block boundary, which is given in the module directory. The length is also given in the directory.

EXT Table (External Symbol Table)

The EXT table contains records (Pascal strings), each of which is the name of a symbol referenced in this module, but not defined in it (i.e., these symbols are declared in another module which this one imports and to which this module is linked at load time).

There may be one EXT table per module. The EXT table begins on a block boundary which is specified in the directory for the module. Its length is also given in the directory. The EXT table is contiguous over its length, which means that individual EXT records within the table may cross block boundaries.

Each EXT record is a multiple of four bytes long. The first byte of each string is its length (according to the Pascal string type); thus strings may be from 1 to 255 bytes long. If strlen(string) + 1 is not a multiple of 4, then 1 to 3 bytes are added as padding to make the EXT record extend to the proper boundary.

The first eight bytes of the EXT table are reserved. Thus, the first string in the table starts at an offset of 8 from the start of the table.

The EXT table is restricted to 65 532 bytes in length (plus the length of the last string). This is so that any entry in the table can be uniquely referenced by 14 bits; the reference is relative to the start of the table. See the description of the reference pointer.

EXT Record

	left byte	right byte
low	LEN=6	S
	Y	M
	B	O
high	L	padding

This one is 8 bytes long.

The formula is:

EXTsize = len + 4 − (len mod 4)

EXPORT

EXPORT is a reserved word used in the Pascal Module. It is used to declare those procedures, functions, constants, types, and variables that are exported, or made available, to other modules that import the module.

Flags

Flags are used in the DEF table, in REF tables, and in the GVR. Their form is shown below.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Primary Type		Data Size			Patchable	Value-Extend	Long Offset

primary type:	00	absolute	:no REFERENCE POINTERS follow
	01	relocatable	:no REFERENCE POINTERS follow
	10	global	:no REFERENCE POINTERS follow
	11	general	:one or more POINTERS follow
data size:	000	signed byte (8 bits)	−128..127
	001	signed word (16 bits)	−32768..32767
	010	signed long (32 bits)	−2147483648..2147483647
	011	(reserved)	
	100	unsigned byte (8 bits)	0..255
	101	unsigned word (16 bits)	0..65535
	110	(reserved)	
	111	(reserved)	

patchable: Indicates that the linker may patch a location in a TEXT record. Applicable only in a REF record and must be false everywhere else.

value extend:
0 No extension present, assume 0
1 Value extension present. Length is 4 bytes.
Always true in DEF records.

long offset:
0 Use short form (1 byte) of offset field. Value is in the range 0..255 and specifies the total length of the GVR except in REF records.
1 Use long form (3 bytes) and offset field. Value is a 24 bit unsigned number in the range 0..16777215. Applicable only in some REF records.

Note

Data Size should be signed long everywhere except in a REF record.

General Value or Address Record (GVR)

The GVR is a variable length record which is intended to represent any absolute, relocatable, or linkable value.

```
TYPE DATATYPE = (sbyte, sword, sint, fltpt, ubyte,uword);
     RELOCTYPE = (absolute, relocatable, global, general);
GENERALVALUE = PACKED RECORD
  PRIMARYTYPE : RELOCTYPE;      (*allows quick indication
                                     of most common types*)
  DATASIZE : DATATYPE;          (*specifies 1,2 or 4 bytes, signed or not*)
  PATCHABLE,                    (*specifies self relative field in branch*)
  VALUEEXTENDED : BOOLEAN;      (*indicates valueextenion*)
  CASE LONGOFFSET : BOOLEAN OF  (*1 or 3 byte offset*)
    FALSE : (short:0..255);     (*unsigned 8 bits*)
    TRUE : long:0..16777215);   (*unsigned 24 bit value*)
END;
  VALUEEXTENTION = PACKED       (*present if value extended bit above is set*)
               RECORD
    CASE DATATYPE OF
        SBYTE,SWORD,SINT,
        UBYTE,UWORD :     (value integer);
END;
REFERENCEPTR = PACKED RECORD   (*one or more present if type = general*)
  ADDRESS : 0..16383;          (*multiply by 4 to get address of EXT symbol*)
  OP : (ADDIT,SUBIT);          (*add or subtract the modifying value*)
  LAST : BOOLEAN;              (*indicated end of list*)
END;
GVR = CONCATENATION            (*NOTE* This is pseudo pascal*)
  GENERALVALUE;                (*2 to 4 bytes of header info*)
  VALUEEXTENTION;              (*0 or 4 bytes of value*)
  ARRAY[zero or more] OF
     REFERENCEPTR;             (*list of EXT references*)
END;
```

IMPLEMENT

IMPLEMENT is a reserved word used in the Pascal Module. It is used as a flag to indicate the beginning of the module body. It is made up of the reserved word plus a declaration statement. The statement can be either empty or used to declare those constants, variables, procedures and functions used internally by the module. None of this information is available outside the module (unless it is also declared in the module's EXPORT section).

IMPORT

IMPORT is a reserved word used in the Pascal Module. It names the modules whose DEFINE SOURCE sections must be examined by the Compiler in order to resolve references to constants, variables, procedures, and functions exported by the modules. The Compiler uses a module's name in conjunction with names of constants, procedures, and functions declared in the module to generate EXT strings for which the loader will search (and link) at run time.

LIBRARIAN

The Librarian is a subsystem designed to manage HP Series 200 Pascal and Assembler object files, link and unassemble object modules, and create system Boot files. It can merge object files containing object modules and optionally link the object modules together. It is the file named LIBRARIAN in your operating system, which can be changed with the Main Level's What command. It is accessed by pressing L from the Main Command Level.

Library

A library is an object file produced by the Assembler, Compiler, or Librarian. Its purpose is to contain object module(s).

LIBRARY

LIBRARY is a special library file included with your operating system. During the boot process, this file (if on-line) generally becomes the System Library; you can also use the What command at the Main Level to specify any file as the System Library.

Only a few useful object modules are included in the file when you received it. Feel free to examine them with the Librarian. Other object modules are supplied on the LIB: and FLTLIB: discs and may be added to the LIBRARY.

Object File

An Object File is the unit of object code managed by the Librarian. It is made up of a Library directory and one or more object modules. The Assembler generates one object file from each source file assembled; the Compiler also generates one object file per invocation. The Compiler's object file can contain one or more object modules depending upon the source file's construction. If the source file contains a number of compilable modules, that number of object modules will be created in the object file.

Object Module

Each object module is made up of a module directory and a module body. The module body is made up of the following items:

One EXT table	A table of the symbols imported by the module.
One DEF table	A table of symbols exported by the module.
One DEFINE SOURCE Area	A software interface between the module and any program which imports it.
One or more TEXT-Record/REF-Table pairs	A TEXT record consists of the constants and code instructions that make up the program. The REF table is a directory of the symbols used in the TEXT record.

Pascal Module

HP Pascal allows source modules to be compiled separately into object modules. The object modules are generally not executable, but are used to complete other Pascal programs. Examples are given earlier in this chapter.

REF Tables

Each REF table follows a TEXT record and is associated with that TEXT record. The REF table begins on a block boundary, which is specified in the directory for the module. Its length is also given in the directory. The REF table is contiguous over its length.

Each REF record is associated with one object (byte, word or integer) within the preceeding TEXT record. There can be at most one REF record for a given object in the TEXT record. The REF records are ordered within the table according to the TEXT objects they reference.

The offset field specifies which text object is referenced. The first REF record gives an offset from the beginning of the TEXT record. Subsequent REF records give an offset from the object referenced by the previous REF record.

low	flags	offset	Ref record is a GVR.
	offset (low part)		offset, 1 or 3 bytes, indicates next object in TEXT record.
high	Ref pointers		Can include any number of Ref Pointers.

Reference Pointer

Bit 15 thru Bit 2	Bit 1	Bit Ø
Address of an EXT Record Relative to Beginning of EXT Table	Add or Sub	End Flag

A REFERENCE POINTER is the relative address of an entry in the EXT table.

The add or sub flag indicates whether the value of the external symbol is to be added (0) or subtracted (1) from the GVR value in order to obtain the actual value. There may be any number of REFERENCE POINTERS in a GVR, and there may be more that one reference to the same EXT record. There may not, however, be both an add reference and a subtract reference to the same symbol, since these would cancel each other.

The end flag indicates whether there are any more REFERENCE POINTERS in the GVR. (0) indicates more to come, (1) indicates the end.

There are two special cases for the EXT address.

- Address 0 (bit pattern 00000000000000xx) refers to the relocation delta for the current module (i.e. new load address minus the old load address).
- Address 4 (bit pattern 00000000000001xx) refers to the global delta for the current module (i.e. new data address minus old data address)

Address 8 (bit pattern 00000000000010xx is the first valid reference to an external symbol.

There are REFERENCE POINTERS in a GVR only if the primary type field specifies general.

System Library

The System Library is a file that is automatically accessed by the Compiler at compile time and by the linking loader at execution time. Object modules stored in this object file are automatically available to any program importing them.

During the booting process, the LIBRARY file usually gets designated as the System Library; however, you can use the What command at the Main Level to specify any file. See LIBRARY above.

Text Record

A Text record is a contiguous section of code, beginning on a block boundary, which is given in the module directory. The length is also given in the directory. The text record can be any arbitrary data, but is usually the object code produced by the Compiler or Assembler.

The Debugger

Chapter 8

Introduction

The Workstation Pascal System features a programming aid called the Debugger. As you probably have guessed, the major purpose of the Debugger is to make program debugging as painless as possible. You may have already seen a reference to this Debugger when you got this this message:

```
RESTART WITH DEBUGGER?
```

The question is in response to a user program generating *but not trapping* an "exception." You will learn how to answer the question in this chapter.

Here are some of the operations you can perform with the Debugger:

- Step through programs on a procedure, statement, or machine-instruction basis.
- Maintain a record of the statements which have already been executed (in order of execution).
- Examine any memory locations and CPU registers, and display the contents in any of the following formats: binary, octal, decimal, or hexadecimal integer; real number; alphanumeric character; and Assembler language (MC68000) instruction.
- Set up "breakpoints" and "error traps" in the program, optionally displaying helpful information when each is encountered.
- Perform number-base conversions and integer arithmetic calculations.

The main emphasis of this chapter is to describe using the Debugger to debug Pascal programs. Debugging an Assembler-language program is more direct; that information is obtainable from the Debugger Reference Section.

Is the Debugger Loaded?

The Debugger is a very powerful subsystem, because it allows any user to access *everything* in the computer. It is therefore a potentially dangerous feature in the hands of users who don't know how to use it (or who you don't *want* to use it). For this reason, and for space considerations, it is *not* automatically loaded when you boot the system. Therefore, you will need to load the Debugger before attempting to use it. (In previous system versions, it was automatically loaded at boot time, as it was part of the INITLIB file). Loading the Debugger is explained in the following Sample Session section.

A Sample Session

This section describes methods for debugging Pascal programs with the aid of the sample program called DEBUG, supplied to you on the DOC (documentation) disc. The program is given in source-code and object-code form.

- DOC:DEBUG.TEXT – the source-code file
- DOC:DEBUG.CODE – the object-code file

The Example Program

A listing of the program is included here for reference. Note that a Pascal program must contain the $DEBUG ON$ Compiler option if you want to have the ability to halt the program at particular line numbers. The effects of this option are further described in the Compiler chapter.

```
Pascal [Rev 3.0  4/15/84] DEBUG.TEXT            28-Apr-84 14:21:55 Page 1

    1:D         0 $DEBUG ON$ { Enable debugging. }
    2:S
    3:D         0 PROGRAM XYZ (OUTPUT);
    4:D         1    VAR
    5:D     -4  1       i           : INTEGER;
    6:D     -8  1       j           : INTEGER;
    7:D    -16  1       x           : REAL;
    8:D    -24  1       y           : REAL;
    9:D    -25  1       ch1         : CHAR;
   10:D    -26  1       ch2         : CHAR;
   11:S
   12:D         1    PROCEDURE Level_1;
   13:D         2       VAR
   14:D     -4  2          i       : INTEGER;
   15:D     -8  2          x       : INTEGER;
   16:D    -12  2          y       : INTEGER;
   17:S
   18:D         2       PROCEDURE Level_2b;
   19:C         3          BEGIN
   20*C         3             WRITE('Level 2b:  ');
   21*C         3             WRITE('  i=',i:2,'  x=',x:4);
   22*C         3             WRITELN('  ch1=',ch1:1);
   23*C         3          END;
   24:S
   25:D         2       PROCEDURE Level_2a;
   26:D         3          VAR
   27:D    -12  3             i, x, y   : INTEGER;
   28:S
   29:D         3          PROCEDURE Level_3;
   30:C         4             BEGIN
   31*C         4                IF i < 4 THEN
   32:C         5                   BEGIN
   33*C         5                      WRITE('Level 3:   ');
   34*C         5                      WRITE('  i=',i:2,'  x=',x:4);
   35*C         5                      WRITELN('  ch1=',ch1:1);
   36*C         5                      i:= i + 1;
   37*C         5                      Level_3;
   38:C         5                   END;
   39*C         4                IF ch1='a' THEN
   40:C         5                   BEGIN
   41*C         5                      ch1:= 'x';
   42*C         5                      WRITE('Level 3:   ');
   43*C         5                      WRITE('  i=',i:2,'  x=',x:4);
   44*C         5                      WRITELN('  ch1=',ch1:1);
   45:C         5                   END;
   46*C         4             END;
   47:S
```

```
48:C        3          BEGIN
49*C        3             WRITE('Level 2a:   ');
50*C        3             WRITE('  i=',i:2,'  x=',x:4);
51*C        3             WRITELN('  ch1=',ch1:1);
52*C        3             i:= 1; x:= 2; y:= 3;
53*C        3             Level_3;
54*C        3             i:= 4; x:= 5; y:= 6;
55*C        3             Level_2b;
56*C        3          END;
57:S
58:C        2       BEGIN
59*C        2           i:= 0; x:= 0; y:= 0;
60*C        2           WRITE('Level 1:    ');
61*C        2           WRITE('  i=',i:2,'  x=',x:4);
62*C        2           WRITELN('  ch1=',ch1:1);
63*C        2           Level_2b;
64*C        2           Level_2a;
65*C        2       END;
66:S
67:C        1 BEGIN
68*C        1    i:= 10;
69*C        1    x:= 20.0; y:= 30.0;
70*C        1    ch1:= 'a'; ch2:= 'b';
71*C        1    WRITE('Main:       ');
72*C        1    WRITE('  i=',i:2,'  x=',x:2:1);
73*C        1    WRITELN('  ch1=',ch1:1);
74*C        1    Level_1;
75*C        1    WRITE('Main:       ');
76*C        1    WRITE('  i=',i:2,'  x=',x:2:1);
77*C        1    WRITELN('  ch1=',ch1:1);
78*C        1 END.
79:S

No errors. No warnings.
```

Please Participate

You will learn much more about the Debugger if you participate in this sample session. Execute the code file one time before attempting the sample session to see the program's output.

Loading the Debugger

As previously mentioned, the Debugger is not automatically loaded as part of the standard system, so you will need to load it into the computer. You can load the module in either of two ways:

- Execute it using the eXecute command (from the Main Command Level); the program installs itself.
- Add the DEBUGGER module to INITLIB, and re-boot the system; the program is then installed automatically.

Loading the Debugger with the eXecute command allows you to use it until you re-boot the system, at which time you will have to eXecute it again to use it. By adding the module to INITLIB, you give all users (who subsequently boot with this INITLIB file) access to the Debugger. You will not want to use this second method unless you want to give all users access to the Debugger.

Executing the Debugger

First, make sure that the ASM: disc is on-line or that the file is otherwise accessible. Then, from the Main Command Level, press the X key to initiate the eXecute command. The system will prompt you with this message:

```
Execute what file?
```

Respond by entering the specification of the DEBUGGER file; `ASM:DEBUGGER.` will work if you are loading the program from the original disc (remember to type a trailing period to suppress the `.CODE` suffix). The system then loads and executes the program, which installs itself in memory.

Adding the Debugger to INITLIB

Use the Librarian (from the Main Command Level) to add this module to the INITLIB file. In general, the steps can be summarized as follows:

1. Make a back-up copy of INITLIB.
2. Edit the INITLIB file with the Librarian, adding the DEBUGGER module (supplied on the ASM: disc) to the file. The DEBUGGER can be anywhere *after* the modules named KEYS, BAT, and CLOCK; however, it must be *before* the module named LAST. (Editing libraries is described in the Librarian chapter.)
3. Store the new library file (using the Keep command).
4. Remove the INITLIB file on your BOOT: disc, and add your new INITLIB file.
5. Re-boot the system.

After re-booting the system, the Debugger should be in memory.

A Note about Key Notations

Throughout this chapter, you will be shown which keys invoke certain Debugger functions. Since you may have one of three different keyboards connected to your computer, each with a different set of keys, you will need to learn which key to press on your keyboard. Here are examples of keys used to invoke a few functions on the three different keyboards[1].

Desired Function	HP 46020A Key(s)	HP 98203B Key(s)	HP 98203A Key(s)
Pause	Break	PAUSE	PSE
Single Step	(System) f5	STEP	STEP
Slow Step	(System) CTRL-f5	CTRL-STEP	CTRL-STEP
Continue	(System) f4	CONTINUE	CONT

For instance, invoke the Pause function on a 46020 keyboard by pressing the Break key. On a 98203B keyboard, press the PAUSE key. With a 98203A keyboard, press the PSE key.

[1] This discussion only gives a few examples; the Debugger Keyboard section near the end of the chapter describes all key(s).

As another example, suppose that you want to invoke the Single-Step function. On both 98203A and B keyboards, press the STEP key; the label is *on the key itself.* On a 46020 keyboard it will be the System key labeled f5 *on the key*, which is labeled **STEP** *on the screen* while in the System-key mode. (If you are not already in System-key mode, then you will need to press the System key before pressing f5). The same notation is used for the other System keys on the 46020 keyboard; the actual System key (i.e., f1 through f8) is not given in text; the label is instead given. You will need to make the association, which you can easily do by looking at the System-key labels while the Menu is being displayed (press the Menu key to toggle the Menu on and off). If you are not familiar with the System and Menu keys, read the discussion in the *Pascal 3.0 User's Guide.*

The convention used in this manual is to show the 46020 keys first (followed by the equivalent 98203B key in parentheses). For instance, the Break (PAUSE) key invokes the Pause function: on the 46020, it is the Break key; on a 98203B keyboard, it is thePAUSE key. (The 98203A PSE key is not shown, because it is close enough to the PAUSE label that you should be able to easily make the connection.)

Is the Debugger Installed?

Before proceeding, you should verify that the Debugger is currently installed. On a 46020 keyboard, press Break (PAUSE) to pause the system. If a P is displayed in the lower, right-hand corner of the screen, then the Debugger is installed. Press **CONT** (CONTINUE) to resume operation.

If the Debugger is not installed, then pressing Break will do nothing.

Invoking the Debugger

The Debugger is called from the Main Command Level. When the Debugger is invoked, the system will then take steps to determine which program you want to debug. Before invoking the Debugger, let's look at how it determines which program to debug.

Specifying a Program

When the Debugger is invoked, the system will either look for a code file on its own or ask you for the code file's name, according to the following priorities. (If the Debugger is not installed, then the D command is identical to the eXecute command.)

1. If there is currently an object-code workfile, the file is automatically loaded into computer memory. If there is a source-code workfile (but not an object-code file), the system reports that it cannot open the file because it was not found.
2. If there is no workfile, the second check made is for the last file compiled since power-up. If present, that file is then loaded.
3. If neither such file exists, you are prompted for a file name.

If you plan to debug, edit, and recompile a program several times in a session, using a workfile may be the best alternative; you will not have to keep typing in the file name, because the current workfile is the automatic object of those subsystems.

For this session, we will set up a workfile. First, use the Filer's What command to see if there is already a workfile. If it happens to be the DEBUG.CODE file, you need do nothing more (before exiting the Filer). Otherwise, use the Get command to specify the example program as the workfile. Here is the prompt you will see:

```
Get what file?
```

Answer by entering the file specification of the example program. Type:

```
DEBUG
```
(Return) or (ENTER)

The filer responds with something like this:

```
Source and code file loaded.
```

You may now Quit the Filer. Now press the (D) key while at the Main Command Level to invoke the Debugger.

Answering RESTART WITH DEBUGGER?
As mentioned earlier, this prompt is shown any time that a "user" program generates *but does not trap* an exception. Answering "Yes" to this question will also get you into the Debugger; you will effectively be at the same point as if you had used the D command. (If the Debugger is not currently installed or if the program was not compiled with the $DEBUG ON$ option, answering "Yes" will only re-execute the program.)

The Debugger Command Screen

You are now in the Debugger's command screen. This message indicates that the Debugger is ready for further instructions:

```
NOW AT START
>
```

The d shown at the lower, right-hand corner of the screen also indicates that you are currently in the Debugger.

The Debugger prompt is a >. When this screen and prompt are displayed, you can type Debugger commands on the last line with the prompt and cursor. Enter each command by pressing the (Return), (ENTER), or (Select) ((EXECUTE)) key. Note that the **CONT** key resumes normal program execution. When execution of the program is complete, control returns to the Main Command Line.

Single-Stepping a Program

When the Debugger is at this starting point, it is ready to step through your Pascal program one statement at a time; this mode is called Single-Step Mode. (It can also do many other things, which will be discussed momentarily.) In the lower, right-hand corner of the screen, the Debugger also conveniently displays the program line number which contains the *next* statement to be executed. This line number corresponds to the line number given in the Compiler listing of the program.

For instance, when debugging our example program, line number 68 is initially displayed. This is the line that contains the Pascal statement that will be executed the next time you press the **STEP** key. Press the **STEP** key once and note that the line number changes to 69, which is the line number of the next statement to be executed.

Pressing **STEP** a second time results in no change in line number. This response is due to the fact that the Debugger steps through the program one *statement* at a time, not one *line* at a time. Pressing **STEP** a third time changes the line number to 70.

Slow Program Execution

The Debugger also allows you to execute a program at a rate of about two statements per second. Press CTRL **-STEP** to use this execution mode (Slow-Step Mode). Line numbers are flashed on the screen as each is encountered. You can return to Single-Step Mode by pressing the **STEP** key.

Returning to the Debugger Command Screen

You may have noticed that the Debugger prompt disappeared when you began stepping through the program. Instead, the Debugger displays the screen that will be used for the program's output so that you can see what the program is doing at each step of execution.

Note that any keys pressed while in this mode appear in the system's type-ahead buffer, not in a Debugger command line. This action allows you to type in responses to any input statements in the program as you would normally type them in. The program reads this buffer when an input statement is encountered and executed.

At some point in the program's execution you may want to to return to the Debugger command screen to execute a command. To do so, press **CRTL-**Break (**CRTL-**PAUSE). The Debugger restores the last Debugger command screen, which is the one that you saw before you began single-stepping the program. You can then execute Debugger commands or return to the program screen by stepping through the program with the **STEP** key.

Toggling Between Screens

While in the Debugger command mode, you can also toggle between these displays (without changing modes) by pressing CTRL **-ALPHA**. For example, suppose you want to quickly check the program screen to see last line displayed by the program. You can do so (without getting out of the Debugger command mode) by pressing CTRL **-ALPHA**. When you've examined all you want on the program screen and are ready to return to the Debugger's command screen, press CTRL **-ALPHA** again.

Screen Dumps

While in the Debugger, you can dump the current contents of the alpha or graphics screens. Use either the **DMP A** (DUMP ALPHA) key or **DMP G** (DUMP GRAPHICS) key, or execute a DA or DG command.

Note that this feature is only allowed when running a program in the processor's "user" state[1]. It is not possible while executing programs in "system" state. If attempted while disallowed, no dump is performed and the following message is displayed:

```
NOT ALLOWED NOW
```

A Look at the Queue

At this point, you may want to continue stepping through the program and noting the order of execution of lines. You can also get a log of all Pascal program lines executed thus far by the Debugger by executing the Queue command. (Actually, these are the line numbers of Pascal statements executed thus far.) Here is an example of the results of this command (assuming that we have only pressed the **STEP** key three times in our example):

```
>Q

206144^   69
206160^   69
206176^   68
206188^   67
START
```

The line numbers are shown in the right column. (The six-digit numbers in the left column, each followed by ^, are memory addresses for use when debugging Assembler language programs; you don't usually need to be concerned with them while debugging Pascal programs.)

Note that the line numbers in the queue are in *reverse order of execution*: the first line executed is at the bottom of the queue listing, the second is listed above the first, and so forth. Also note that the line at the top of the list has *not* yet been executed; it will be executed the next time the **STEP** key is pressed.

Note that Pascal line numbers will *only* be shown if the $DEBUG ON$ Compiler option was used.

Note also that when the question `RESTART WITH DEBUGGER?` is displayed after encountering an exception that was not trapped, you can get a listing of the queue by pressing CTRL-Break and then executing a Q command. You can also direct the Debugger to trap exceptions, as described in the Exception Trapping section of this chapter.

Displaying Data

Before showing how to use many of the more powerful Debugger features, let's look at some simple Display operations. Execute the following command:

```
>D 8+32
           +40
```

[1] All user programs are executed in the "user" privilege state, while system programs, such as the Editor, Filer, and so forth, are executed in "supervisor" privilege state. See the *MC68000 User's Manual* for a more comprehensive description of these states.

From this example, you can deduce that the literal numbers that you entered were interpreted as decimal integers and that the result was a signed decimal integer.

Note that you don't need to specify the `D` in commands that begin with non-alphabetic characters.

```
>8+32
        +40
```

Now execute this command:

```
>D -32768-32768
       -65536
```

From this result, you can see that the range of integers is at least 16 bits. In fact, it is 32 bits, which indicates that a four-byte register could be used to store the numbers and results. The range is -2^{31} through $2^{31} - 1$ (or 2 147 483 648 through 2 147 483 647).

Executing these commands might help to see this range of integers more clearly:

```
>D 127*256*256*256
+2130706432
>D 128*256*256*256

OVERFLOW
```

Note that *only* integer arithmetic operations are performed. For instance, division produces only the dividend, not the remainder:

```
>4/3
         +1
```

Display Formats

Since the Debugger uses the processor's 32-bit registers for expression evaluation, most results are formatted using four-byte quantities. Here are two equivalent examples of using the default format of one signed (four-byte) INTEGER:

```
>D 255
        +255
>D 255:1I4
        +255
```

The format specifier is the ':1I4' appended to the literal number 255: the leading `1` indicates that one quantity is to be generated; the `I` indicates that the quantity is to be displayed as a signed, decimal Integer; the trailing `4` indicates that 4 bytes are to be formatted.

If the default format of one four-byte decimal integer is not what you'd like, you can explicitly specify another format. For example, the following command generates four one-byte Binary numbers (the `!`'s indicate binary notation):

```
>D 1024+255:4B1
!00000000 !00000000 !00000100 !11111111
```

Here is an example of formatting the integer into four one-byte octal numbers (the %'s indicate Octal notation):

```
>D 1024+255:4O1
%0 %0 %4 %377
```

Now specify that the number is to be formatted as one four-byte hexadecimal number with either of the equivalent commands (the $'s indicate hex notation):

```
>D 1024+255:1H4
$000004FF
>D 1024+255:H
$000004FF
```

The leading 1 and trailing 4 are the defaults assumed when these parameters are omitted.

This format specification directs the Debugger to display two bytes as a hex value:

```
>D 1024+255:H2
$0000
```

Note that 0's were displayed because the Debugger begins with the *most*-significant bits of the four-byte integer. Here is a more meaningful display format for the same data:

```
>D 1024+255:2H2
$0000 $04FF
```

It is also possible to display literal strings with the data you are formatting for the display. Either single or double quotes can be used to delimit the string. For example, this command gives a more descriptive display:

```
>D -7:'-7 in Hexadecimal = ',H4
-7 in Hexadecimal = $FFFFFFF9
```

You can also disassemble machine-language instructions by using the X format specifier. Here is an example:

```
>D $4E750000:X
RTS
```

This is usually only helpful while debugging Assembler-language programs. (Note that you must load module REVASM into memory with the P-load command from the Main Command Level in order to use this format.)

Another format specifier is the slash (/). When a "/" is encountered in a Display command, the display is continued on the next line.

```
>D 23+45:/,'RESULT = ',I4/

RESULT =           +68

>
```

Input Formats

The !, %, and $ symbols preceding numbers in the above examples were used to indicate the base of the numbers displayed on the screen. Similarly, you can use them with literal numbers input in the command. This feature allows number-base conversions.

For example, suppose that you want to convert the binary number 11001010 to its decimal representation. Here is a sample command:

```
>D !11001010:I
         +202
```

To convert the number to hex, execute this command:

```
>D !00110110:H
$00000036
```

Changing the Default Display Format

The default format can be changed by giving an F (Format) command. For example, the following command changes the default to ':1H4', which instructs the Debugger to take 4 bytes and display them as one four-byte Hexadecimal value:

```
FH
```

This command sets the default format to Octal (':1O4'):

```
FO
```

This command sets the default format to Binary (':1B4'):

```
FB
```

This command changes the default format to ':1U4', which directs the Debugger to display 1 four-byte Unsigned decimal integer.

```
FU
```

This command sets the default format back to signed decimal Integer (1I4):

```
FI
```

Now that you've had an introduction to the Display commands, let's look at some more powerful commands.

Controlling Execution with Breakpoints

A breakpoint is a point in the program where you want execution to be temporarily halted. With a Pascal program, the point will be at a program line. Thus, when the Debugger is executing a program and encounters a breakpoint, it halts just *before* executing the program line.

Setting Breakpoints

To set a breakpoint, use the BS command. Specify the location as an integer which follows the letters "BS", separated by a space. For example, to set a breakpoint at Pascal program line 74, enter the following command:

```
BS 74
```

Press **CONT** (**CONTINUE**) and the program begins executing again. When it encounters line 74, it pauses before executing the line and displays the message:

```
NOW AT LINE 74
```

The Debugger then prompts you for another command. At this point, you can do any of the following:

- Step through the program one line at a time (if it was compiled with $DEBUG ON$)
- Execute other Debugger commands (such as examine memory or register contents)
- Continue the program

Once the program has finished execution, all breakpoints are automatically de-activated. You will have to explicitly re-activate them, as described in a subsequent section.

Up to nine such breakpoints may be defined at one time. Most breakpoints remain in effect until cleared or de-activated.

The Count Option

An optional count can be included by adding an integer after the location. The count instructs the Debugger to stop when it reaches the location the indicated number of times. For example, enter the following command:

```
BS 31 3
```

This particular command instructs the Debugger to halt the program immediately before the third execution of line 31. Press **CONT**, and the program executes until line 31 is reached the third time and then halts. Note that this type of breakpoint is automatically cleared when encountered the specified number of times.

Breakpoints with Commands

Another form of the BS command is the "BS" and the location number followed by a Debugger command string enclosed in quotes. The command string is one or more legal Debugger commands (separated by semi-colons). These commands are immediately executed when the location is encountered. The Debugger automatically continues program execution after executing the command string. Here is an example that will provide a visual record of how many times that line 37 was executed:

```
BS 37 "D 'LINE 37 REACHED.'"
```

Of course, you will need to get back into the Debugger command screen to see the results of this breaknoint being encountered. The D (Display) command is explained in detail later.

You can alternately pause the program by making the last command in the string a question mark. This command directs the Debugger to pause and wait for input from the keyboard. For example, enter the following command breakpoint:

```
BS 59 "D PC; ?"
```

The Debugger stops at line 59, displays the Program Counter, and waits for input.

Here is another example of using a breakpoint with a command:

```
BS 59 'IF 1=1; D "1=1"; ELSE; D "1<>1";?; END'
```

The relational expression following the IF command, in this case 1 = 1, is first evaluated. If it is true, then the command(s) between the IF and the ELSE are executed. If it is false, then the command(s) between the ELSE and END are executed. This type of command is useful for purposes like checking the value of a variable and then pausing if its value is out of an expected range. The IF, ELSE, and END commands are further explained in the reference section. Checking the value of variables is explained later in this tutorial.

Deactivating Breakpoints

The BD command deactivates breakpoints. If a line number is included, the breakpoint is deactivated for that line number. For instance, the following command deactivates the breakpoint at line 41:

```
BD 41
```

If no line number is included, all breakpoints are disabled. For example, this command disables all breakpoints:

```
BD
```

Displaying the Breakpoint Table

The B command displays the breakpoint table or the one at the specified line number. Execute the following command:

```
B
```

and you'll see a display similar to the following:

```
>B
BREAK PDINTS
------------
A          74          0
A          37 D 'LINE 37 REACHED'
A          59 D 'IF 1=1;D "1=1";ELSE;D "1<>1";?;END
```

The first character on each line of the table is either "A" for active, or "D" for deactivated. The second parameter is the line number of the breakpoint. If the third entry in the table is a positive number, then a count option is in effect for the breakpoint (execution will pause when the Debugger reaches the line that number of times). If the third entry is a command string, then that command is executed each time the line is encountered. If it is a "0", then it is a normal breakpoint (i.e., no count nor command was specified with the breakpoint).

Reactivating Breakpoints

The BA command reactivates disabled breakpoints. If the line number is included, the breakpoint is reactivated for that line number, otherwise, all breakpoints are affected. For example, the following command reactivates the breakpoint that was deactivated in the example above:

```
BA 41
```

Try the B command to see the table again.

When a program runs to completion and is then restarted (by pressing the D key), the breakpoints are still there; they are just deactivated. Use the BA command to reactivate some or all breakpoints.

Clearing Breakpoints

The BC command clears breakpoints by removing them from the table. If a line number is included, the breakpoint is removed *only* for that line; otherwise, all breakpoints are cleared. Enter the following command to remove only the breakpoint at line 41:

```
BC 41
```

The Pause Function and Breakpoints

If the Debugger is not installed, the Break (PAUSE) key is a no-op. The rest of this discussion assumes that the Debugger is installed.

While *not* in the Debugger command mode, pressing Break effectively halts any program at the current execution point. (Note that this key may not pause the program on a line boundary like the **STEP** key does.)

While *in* the Debugger command mode, however, pressing Break returns you to the user program display and pauses the program at the current execution point. Press continue to finish program execution.

If you press Break after encountering an active breakpoint, it will also get you to the program's display. However, if you pause exactly on a currently active breakpoint (but before encountering it), pressing Break will not get you into the program's display. You would have to press Break again cause the breakpoint to take immediate effect. **CONT** will then work as expected.

Executing a Number of Statements

Go commands set a tenth temporary breakpoint. They are one-time commands to pause execution before a specified program instruction.

The G command tells the Debugger to Go. If you include a number after the "G", that number of statements is executed, after which the Debugger halts and waits for another command. For example, this command tells the Debugger to Go 8 statements:

```
G 8
```

If no number is given, the remaining instructions are executed (same as pressing **CONT**).

The GF (Go and Flash) command is the same as the G command except execution is slowed and line numbers are flashed in the lower right corner of the screen.

The GT (Go 'Til) command is the same as Go except a location is specified rather than a count. For example, this command tells the Debugger to Go 'Til line 39 is reached:

```
GT 39
```

Another form of this command tells the Debugger to Go 'Til the location is reached a number of times. For example, the following command tells the Debugger to stop before line 41 is executed the third time.

```
GT 41 3
```

The GT statement also allows the command string option. For example, this command directs the Debugger to do the following: execute the program until line 42 is reached, then display the Program Counter and await further instructions.

```
GT 42 "D PC; ?"
```

The GTF (Go 'Til and Flash) command is the same as GT except execution is slowed and line numbers are flashed in the lower right corner of the CRT.

Tracing Program Flow through Procedures

You can also halt execution of a program as it enters and exits procedures. For instance, suppose that you want to halt the program when the current procedure is exited. To do that, execute the PX (Procedure eXit) command:

```
PX
```

Execution will be halted after the procedure is exited (i.e., after the last line of this procedure is executed, but before the subsequent program line is executed). For instance, executing this command while in Level_3 results in this display:

```
 >PX
PROC EXITED
```

The Debugger shows that the next line to be executed is line 31.

To halt the program at the point that the current procedure (or main program) calls another procedure, use the PN (Procedure Next) command:

```
PN
```

When the next procedure is encountered, the Debugger reports this message:

```
NEXT PROC
```

and the program is halted before executing the first executable line of the procedure. If the current procedure is exited before another is called, the Debugger reports this message:

```
PROC EXITED
```

A Look at the Stack Frame

Another handy feature to use while walking though the program on a procedure basis is the SF (Stack Frame) command. Here is an example display of this command:

```
>SF
          59  LEVEL_1
PROC ADDRESS       -403316^
CALLED FROM        -402718^
      LINE              74
```

The first line of the display shows the first (executable) line of the program next to the procedure's name. The second line shows the memory address of the procedure (which is not important while debugging Pascal programs). The third line shows the address of the procedure. The fourth line shows the number of the line from which this procedure was called.

Examining Variables

Without the ability to check the value of program variables, debugging a program could become more tedious than it already is. Rest assured that this Debugger does allow you to look at the contents of any variable in computer memory. However, in order to check the contents of program variables, you will need to know two important facts: where they are in memory, and how to format them into an understandable form.

To see where a variable is stored in memory, it is necessary to look at the Compiler listing. Each variable has a negative integer printed next to it on the listing. This negative value is the offset (in bytes) from the base address where the variables are located. The base address for a procedure's local variables is the current stack frame pointer (SF); the main program's variables have a base address offset the value of the program name (here XYZ) from A5.

That's why it's helpful if, when writing the program, you declare each variable on a separate line so that an offset will be printed on the listing for each variable. Alternatively, you can use the $TABLES$ Compiler option to get a printout which tells all about each data type and variable. This option is explained further under "Structured Variable Formats".

To format the variable's value in memory, you will need to use the Display command. Let's go back to the example program and let it finish by clearing the breakpoints using "BC" and then press **CONT**. Restart the program and then execute `GT 60` command to Go Til line number 60.

To see the value of the local variable i that is declared in the procedure called Level_1, look at the Compiler listing (line 14) to see that it has an offset of −4. This is an offset from the stack frame pointer (SF) of that procedure. Subtract 4 from the stack frame pointer, and use "^" after the expression to indicate you want the contents of the memory location *referenced by* the value of the expression in parentheses. Enter the following command:

```
>D (SF-4)^:'   X = ',I
  X = 0
```

To see the value of y, execute:

```
>D (SF-8)^:'   Y = ',I
  Y = 0
```

And to see the value of z, execute:

```
>D (SF-12)^:'  Z = ',I
  Z = 0
```

You may also specify that all three integer variables be displayed at the same time by executing this command:

```
>D (SF-12)^:3
0       0       0
```

The display will show the three integer variables separated with spaces. The variable with the offset of −12 will be the first one displayed, the one with the offset of −8 is second, then the third one has offset −4.

When looking for local variable values, be sure that you have stopped the program in the procedure that defines the variables. Each procedure that is called has a stack frame created for it even if there are no local variables. If you have stopped the program in a procedure which is contained inside of another procedure, you can use the walk commands to get to the stack frames of the outer level procedures (see "Static and Dynamic Links").

The global variables in the main program or globals declared in modules are located at offsets from their specific global area. The respective areas have a symbol associated with each one. The symbol has a value which is equal to the offset or distance from (A5). So when you reference these variables, add the program or module name to A5 and then subtract the offset for the particular variable location.

For example, if you wanted to see the value of the variable x in the main program (here it is named XYZ), use this command:

```
D (A5+XYZ-4)^
```

To see the value of Y, execute:

```
D (A5+XYZ-8)^
```

To see the value of the two character variables in ch1 and ch2 (of program DEBUG), it is necessary to specify a format, because the default format is integer. To see the variables ch1 and ch2, execute this command:

```
D (A5+XYZ-26)^:2A1
```

The format specifies that 2 Alpha values are to be displayed, each having 1 byte. They are located at an offset of -26 from the value of symbol XYZ, relative to A5.

The processor registers that can have their values displayed are listed below:

```
A0..A7          (the Address registers)
AA              (All Address registers)
D0..D7          (the Data registers)
DD              (all Data registers)
PC              (the Program Counter)
SR              (the Status register)
```

To display the numeric values of the contents of address register A0 and the Program Counter, execute this command:

```
D A0 PC
```

To display the numeric value at the location *referenced* by the the Program Counter (i.e., whose address is stored in the PC), execute the following command:

```
D PC^
```

To display the value at the location *referenced* by the Program Counter, interpreting it as an Assembler language instruction, execute this command (remember that module REVASM must be P-loaded to use this format):

```
D PC^:X
```

The Debugger symbols and corresponding definitions are as follows:

```
LN (Line Number)
EC (Escape Code)
IO (I/O result code)
GB (the Global variable Base)
RB (the code Relocation Base)
SF (the current Stack Frame pointer)
```

Examining Consecutive Memory Locations

The Open command is like the Display command except the address is displayed with the value and you are prompted to press either the up-arrow key or the down-arrow key. This causes the address value to increment or decrement depending on the key choice. The adjustment is 1 byte with the OB command, 2 bytes with the OW command and 4 bytes with the OL command. When you have seen enough, press (Return), (ENTER), or (Select) ((EXECUTE)) to terminate Open mode and return to the Debugger command mode. For example, to see the hex values which are the machine codes for the current program, use this command:

```
FH
```

(See the Default Formats section for more details.)

To examine (16-bit) word pointed to by the current contents of the Program Counter, use this command:

```
OW PC^
```

The > to the right of the display prompts for an up-arrow key (↑) or down-arrow key (↓). To see the next word in memory, press the up-arrow key. Continue until you have seen enough. Press (Return) or (ENTER) to exit the Open command.

Formats for Structured Variables

There is a mechanism for displaying non sequential values also. It is necessary to specify one memory location to set the memory pointer. Then by using special symbols, you can alter the value in the memory pointer. You can also display the value of the memory pointer. All these symbols are part of the format and are typed following the location specification and a colon (:).

```
"*" is the value of the memory pointer

"<" preceded by a number, decrements the value of the memory
    pointer by the number

">" preceded by a number, increments the value of the memory
    pointer by the number

"^" causes the memory pointer to take the value at the
    location indicated by the current pointer
```

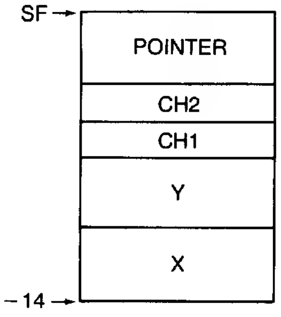
These mechanisms make it possible to examine different fields of structured variables.

First, a note about structured variables. When space is allocated for a structured variable, the number of bytes needed is determined and given to the variable. The individual elements of the structure are then assigned space at ascending locations. For example, if you had the following Pascal record, 14 bytes are needed to store the whole record:

```
Pasc_Rec = RECORD
               x : INTEGER;
               y : INTEGER;
               ch1 : CHAR;
               ch2 : CHAR;
               pointer : ^Pasc_Rec;
           END;
```

If a variable of this type is the first variable for a procedure, then the record would occupy the first 14 bytes below the stack frame pointer (SF-14)^. The elements in the record would be at positive offsets from this location. Variable x would have an offset of 0 (SF-14)^; y has an offset of 4 (SF-14+4)^; Ch1 has an offset of 8 (SF-14+8)^; etc. This information is easily obtainable when the $TABLES$ Compiler directive is used.

The following drawing illustrates the structure of the RECORD variable in memory.



Rather than displaying the values of the record individually, you can use the following Debugger command:

```
D (SF-14)^:I4,4>,ZA1,^,4>,I4,*
```

This command tells the Debugger to go to the memory location 14 bytes below the Stack Frame pointer (the bottom of the record), display the four-byte integer (x), go up 4 bytes and display the 2 Alpha characters, assume the value that is stored after the characters (the pointer field), then go up 4 bytes in the new record and display the four-byte integer (y), and then display the current location. Notice that the Debugger display pointer is left at the subsequent locations after the particular displays are made. In other words, after the display of (x), it is only necessary to move 4 bytes rather than 8, to position the display pointer to the character variables.

Changing Memory Contents

The ability to change the values in memory is, among other things, the ability to get a program back on the right track. In one Debugger session, you can detect several problems with a program without having to stop, edit and recompile the program for each one. Simply change the values of the variables that are causing the problem. To change the values of variables in a Pascal program, use the Open commands. Variables are referenced the same way they are with the Display command.

The Open commands are as follows:

- OB - for byte values.
- OW - for word values.
- OL - for long word (four-byte) values.

Suppose you want to change the value of a variable to 8; assume that it is local to the current procedure, that it is an integer variable, and that it has an offset of −4 from the procedure's stack frame pointer (SF). It is necessary to use the OL form of the Open command, since integers are 4 bytes long. Execute the following command:

```
OL (SF-4)^ 8
```

As another example, suppose you want to change the value of the global (main program's) variable ch1 to "x". Because characters only use 1 byte of storage, use the OB form of the command.

```
OB (GB-25)^ "x"
```

By changing the values of those variables, the sequence of execution is drastically altered.

Static and Dynamic Links

Each time a procedure is called in a Pascal program, a new stack frame is created. This stack frame contains all the local variables in the procedure as well as the procedure's static and dynamic links[1].

The Debugger contains a mechanism for following these links. It is the Walk command. The Walk command takes three forms:

- WS - follows the static link back one step.
- WD - follows the dynamic link back one step.
- WR - resets to the current stack frame.

There are no options or parameters. These commands in no way affect or influence program execution.

Restart the Debugger by pressing the Stop key and the D key. Set a breakpoint on line 30 for the third execution of the procedure Level_3.

```
BS 30 3
```

Press Return or ENTER, and then press **CONT**. The program will stop the third time line 30 is reached.

The sequence of calls is as follows:

```
Program XYZ

Procedure Level_1

Procedure Level_2b

Procedure Level_2a

Procedure Level_3

Procedure Level_3

Procedure Level_3
```

Give six successive WD (Walk Dynamic) commands and you'll get the above information presented in reverse order. The information displayed for each WD command is the stack frame information for the current procedure and then the same for the calling procedure. The stack frame pointer is updated to point to the calling procedure's stack frame. You can look at those variables and the links stored in that stack frame. Consecutive WD commands walk us back through the entire calling sequence. We can stop anywhere along this path and examine the variables in a procedure's stack frame.

[1] Static and dynamic links are described in detail in the section of the Compiler chapter called How Pascal Programs Use the Stack.

To return to the stack frame for Level_3 where you stopped the program, execute:

```
WR
```

This command resets the Debugger stack frame pointer variable.

You can also walk the static link. This gives you the ability to examine variables whose scope includes the current procedure. Type:

```
WS
```

This command brings us to the Stack Frame for Level_2a which contains the variable x.

Use the Display command to examine the value of x.

```
D (SF-4)^
```

The value of x is displayed.

The value of x is only affected by successive executions of Level_3. If Level_3 had local variables, they would display different values in each stack frame. However, only one copy of the variable x exists in the one stack frame for procedure Level_2a. The value of x is as it was when we stopped program execution during the third invocation of Level_3. That value is 3.

Exception Trapping

It is possible to stop execution of a program at an exception to normal processing. Normally, an escape is made by the system and successive recovery mechanisms allow termination of the program. At the time of termination, the system displays the escape code and the line number in the outer level recovery. The escape code is valid information, but the line number may not be the location of the error. By re-executing the program with a trap set for the exception, we can stop execution at the point of the error, have the actual line number of the error displayed, and examine variables for the problem.

There are three commands for exception trapping. We can trap selected escape codes with the Escape Trap instruction. The following command directs the Debugger to trap only escape code 100.

```
ET 100
```

When escape code 100 is encountered, control is returned to the Debugger and the following message is displayed on the screen:

```
-EXCEPTION-
ESCAPE CODE 100
SR=$0000 PC=    -207532 LINE        +12
```

We can stop at all *except* selected escape codes with the Escape Trap Not instruction. This command directs the Debugger to trap every escape code except 100.

```
ETN 100
```

Not specifying an escape code causes the command to work for every escape code. This command directs the Debugger to trap *all* escape codes.

```
ET
```

This command doesn't trap any exceptions.

```
ETN
```

When the exception occurs, execution stops and control is transferred to the Debugger. At that point, you can examine the state of the program.

When the Debugger is initiated, the default escape trapping command is the following:

```
ETN 0 -20
```

These are the escape codes for normal termination and the Stop key. The Debugger will trap all escape codes except those.

The third type of escape trap command allows you to execute command(s) when the escape is detected. Here is an example:

```
ETC 'D "ESCAPE HAS OCCURRED";?'
```

This command displays its message and then halts the program, awaiting further Debugger commands.

Generating Escapes

With the Debugger, you can also generate escapes. For instance, this command generates an ESCAPE(10) at the current point in the program.

```
>EC 10
```

The result of this command is the same as if the program had encountered the escape at the current location. If you have an ET command currently defined for the escape code, the Debugger will trap it also.

A Note about Assembly Language Programs

All of the Debugger commands apply when debugging an Assembler language program as well. The difference is that the location specification is given as an address and not a line number. An address is specified with a "^" appended to the location specifier. For example, the following command says to Go Til the address 1423 is encountered:

```
GT 1423^
```

The Debugger knows about symbols which have been DEFed. The entry points into assembly modules, programs, and procedures should have been defined (with DEF). You can specify an address in an assembly routine by specifying an offset from the routine's entry point. The offset in the routine can be found on the Assembler output. For example, the following (equivalent) commands direct the Debugger to Go Til encountering the the address 16 decimal (10 hex) memory locations past the entry point into "routine":

```
GT (routine+16)^
```

or:

```
GT (routine+$10)^
```

Read about the particulars of each command in the subsequent Command Reference section.

Debugger Keyboard

This section describes the key definitions while in the Debugger. Note that once you are in the Debugger there are two modes: Command Mode and Step Mode.

A Note about Key Notations

Throughout this section, you will be shown which keys invoke certain Debugger functions. Since you may have one of three different keyboards connected to your computer, each with a different set of keys, you will need to learn which key to press on your keyboard. Here are examples of keys used to invoke a few functions on the three different keyboards.[1]

Desired Function	HP 46020A Key(s)	HP 98203B Key(s)	HP 98203A Key(s)
Pause	Break	PAUSE	PSE
Single Step	(System) f5	STEP	STEP
Slow Step	(System) CTRL-f5	CTRL-STEP	CTRL-STEP
Continue	(System) f4	CONTINUE	CONT

For instance, invoke the Pause function on a 46020 keyboard by pressing the Break key. On a 98203B keyboard, press the PAUSE key. With a 98203A keyboard, press the PSE key.

As another example, suppose that you want to invoke the Single-Step function. On both 98203A and B keyboards, press the STEP key; the label is *on the key itself.* On a 46020 keyboard it will be the System key labeled f5 *on the key*, which is labeled **STEP** *on the screen* while in the System-key mode. (If you are not already in System-key mode, then you will need to press the System key before pressing f5). The same notation is used for the other System keys on the 46020 keyboard (i.e., f1 through f8): the actual System key is not given in text; the label is given instead. You will need to make the association, which you can easily do by looking at the System-key labels while the Menu is being displayed (press the Menu key to toggle the Menu on and off). If you are not familiar with the System and Menu keys, read the discussion in the *Pascal 3.0 User's Guide.*

The convention used in this manual is to show the 46020 keys first (followed by the equivalent 98203B key in parentheses). For instance, the Break (PAUSE) key invokes the Pause function: on the 46020, it is the Break key; on a 98203B keyboard, it is the PAUSE key. (The 98203A PSE key is not shown, because it is close enough to the PAUSE label that you should be able to easily make the connection.)

Is the Debugger Installed?

Before proceeding, you should verify that the Debugger is currently installed. Press Break (PAUSE) to pause the system. If a P is displayed in the lower, right-hand corner of the screen, then the Debugger is installed. Press **CONT** (CONTINUE) to resume operation.

If the Debugger is not installed, then pressing Break will do nothing.

[1] This discussion only gives a few examples; the Debugger Keyboard section near the end of the chapter describes all key(s).

Calling the Debugger from the Main Command Level

D

From the Main Command Level, pressing the D key calls the Debugger (if installed).

Step Modes

Here are the available operations and key definitions while in the Debugger Single-Step and Slow-Step Mode.

Getting into the Step Modes

STEP

Causes the program to halt on the next line number; or, if already halted, execute one Pascal statement. (This key gets you into the Single-Step Mode.)

CTRL **-STEP**

Causes program execution to be slowed (to about 2 statements per second) and line numbers displayed. (This key gets you into the Slow-Step Mode.)

Controlling Program Execution

Break
(PAUSE)

Program execution is paused. Note that the type-ahead buffer is still active and immediate-execute keys still function (e.g,. **DMP A**).

Stop

Stops program execution.

Getting into Command Mode

CTRL-Break
(CTRL-PAUSE)

This key provides immediate entry into Debugger Command Mode.

Returning to the Main Command Level

CONT
(CONTINUE)

Causes program exection to resume with Step mode cancelled.

Command Mode

Here is a description of available operations and key definitions while in the Debugger Command Mode. If it is not installed, the command is identical to the eXecute command.

Entering Commands

Alphanumeric Keys

Used to enter Debugger commands. The characters generated are uppercase; you must use Shift to produce lowercase characters.

Return or ENTER

Terminates input and initiates execution of the command.

Select
(EXECUTE)

Terminates input and initiates execution of the command.

CTRL -**ALPHA**	Alternates between the Debugger command screen and System screen (CTRL-EXEC on the 98203A keyboard).
Caps	Undefined (keyboard is always in CAPS mode).
Shift with numeric-pad keys	Produces special characters (only 46020 keyboards).
CTRL with alphanumeric and numeric-pad keys	Allows entry of ASCII control characters.
Backspace	Back space the cursor and blanks one character (If the cursor is in the extreme left, this key is a no-op).
Clear line or Delete line (CLR LN or DEL LN)	Clears the input line.
RECALL	Clears the input line and recalls the last executed line.
Insert char (INS CHR)	Inserts one (1) blank character at the cursor position (does *not* switch to an "insert mode," as there is none).
Delete char (DEL CHR)	Deletes the character at the cursor position.
CLR-END (CLR→END)	Deletes all characters to the right of the cursor.
f1 thru f8 (k0 thru k9)	Typing-aid keys (explained under K commands).
Knob	Same as left/right arrow keys.
Shift-Knob	Same as up/down arrow keys.
Left-arrow and Right-arrow	Move the cursor in the corresponding direction.
Up-arrow and Down-arrow	Have meaning only with the Open commands (OL, OW, OB).

Screen Control

Clear display
(CLR SCR)
Clears the alpha raster. In Step Modes, this key clears the System screen; in Command mode, it clears the Debugger Command screen.

ALPHA
Turns on the alpha raster and turns off the graphics raster (SHIFT-RCL on the 98203A keyboard).

DMP A
(DUMP ALPHA)
Performs a DUMP ALPHA function (the current alpha raster is sent to the PRINTER: volume). (SHIFT-INS C on the 98203A keyboard).

CTRL **-ALPHA**
Alternates between the Debugger and System screen images (CTRL-EXEC on the 98203A keyboard).

GRAPH
(GRAPHICS)
Turns on the graphics raster and turns off the alpha raster. (SHIFT-INS L on the 98203A keyboard.)

DMP G
(DUMP GRAPHICS)
Performs a DUMP GRAPHICS function (sends the current graphics raster to the PRINTER: volume). (SHIFT-DEL C on the 98203A keyboard.)

Controlling Program Execution

Break
(PAUSE)
Program execution is pauseed. Note that the type-ahead buffer is still active and immediate-execute keys still function (e.g,. **DMP A**).

Stop
Stops program execution.

Getting into a Step Mode

STEP
Causes the program to continue executing until the next line number is encountered (i.e., gets you into Single-Step Mode).

CTRL **-STEP**
Causes the program to continue executing slowly, and line numbers displayed as encountered (i.e., gets you into Slow-Step Mode).

Returning to the Main Command Level

CONT
(CONTINUE)
Causes program exection to resume with Command mode cancelled.

Debugger Command Summary

This section briefly summarizes the Debugger commands for quick reference purposes. A more complete description of each command is presented in the following Command Reference section.

Breakpoint Commands

BS – Sets a breakpoint at the specified location.
BD – Disables (but does not remove) breakpoint(s).
BA – Activates disabled breakpoint(s).
BC – Clears breakpoint(s).
B – Displays the breakpoint table.

Call Command

CALL – Calls the machine language routine at the specified memory address.

Display Commands

D – Displays the specified object(s). Objects can be specified immediately, directly, or indirectly. Formats describe the internal representation of the data.
TD – Displays the command string which is defined by the softkey **k4**.
TD I – Restores the initial command string to **k4**.

Dump Commands

DA – Performs the DUMP ALPHA function.
DG – Performs the DUMP GRAPHICS function.

Escape Code Commands

EC – Generates the specified escape.
ET – Sets up escape trapping of specified escape codes; Debugger halts when an escape is executed.
ETC – Sets up escape trapping of all codes; Debugger executes the specified command when an escape is executed.
ETN – Sets up escape trapping of all codes *except* those specified; Debugger executes the specified command when an escape is executed.

Format Commands

FB – Sets the default display format to Binary.
FH – Sets the default display format to Hexadecimal.
FI – Sets the default display format to signed Integer.
FO – Sets the default display format to Octal.
FU – Sets the default display format to Unsigned integer.

Go Commands

G – Causes execution to resume (same as **CONTINUE**).
GT – Causes execution to resume until specified location is encountered.
GTF or **GFT** – Same as GT except that execution is slowed and the line numbers are flashed in the lower right-hand corner of the screen.

IF, ELSE, and END Commands

IF – Allows conditional execution of subsequent commands based on the result of evaluating the specified expression.

ELSE – Delimits the commands that will be executed when the IF condition is FALSE.

END – Ends the IF command.

Open Memory Commands

OB, **OL**, and **OW** – Used to display (and optionally alter) the values of memory locations.

Procedure Commands

PN – Halts program execution when the next procedure is called (or when the current one is exited, whichever occurs first).

PX, or P – Halts program execution when the current procedure is exited.

Queue Commands

Q – Displays the Queue, which is a record of which line numbers were executed (or PC values of instructions executed).

QE – Ends recording of line number values in the Queue.

QS – Starts the recording of information in the Queue.

Register Operations

A0..A7, D0..D7, PC, SP, US, SR – Display or assign values to the corresponding processor register(s).

Softkey Commands

k0 .. **k9** – Defines the command string to be displayed when the softkey is pressed (while in the Debugger).

System Boot Command

sb – The system boot command puts the computer in the power-up state for re-booting. (The command must be typed in lowercase letters.)

Trace Commands

T – Causes the specified number of instructions to be executed, each followed by an implicit TD command.

TQ – Same as the T command except that the TD command is executed only after the last instruction.

TT – Same as TQ except that a location is specified rather than a count.

Walk Procedure Links Commands

WD – The Stack Frame pointer (SF) is moved to the stack frame of the calling procedure.

WS – The SF is moved to the stack frame of the nesting procedure.

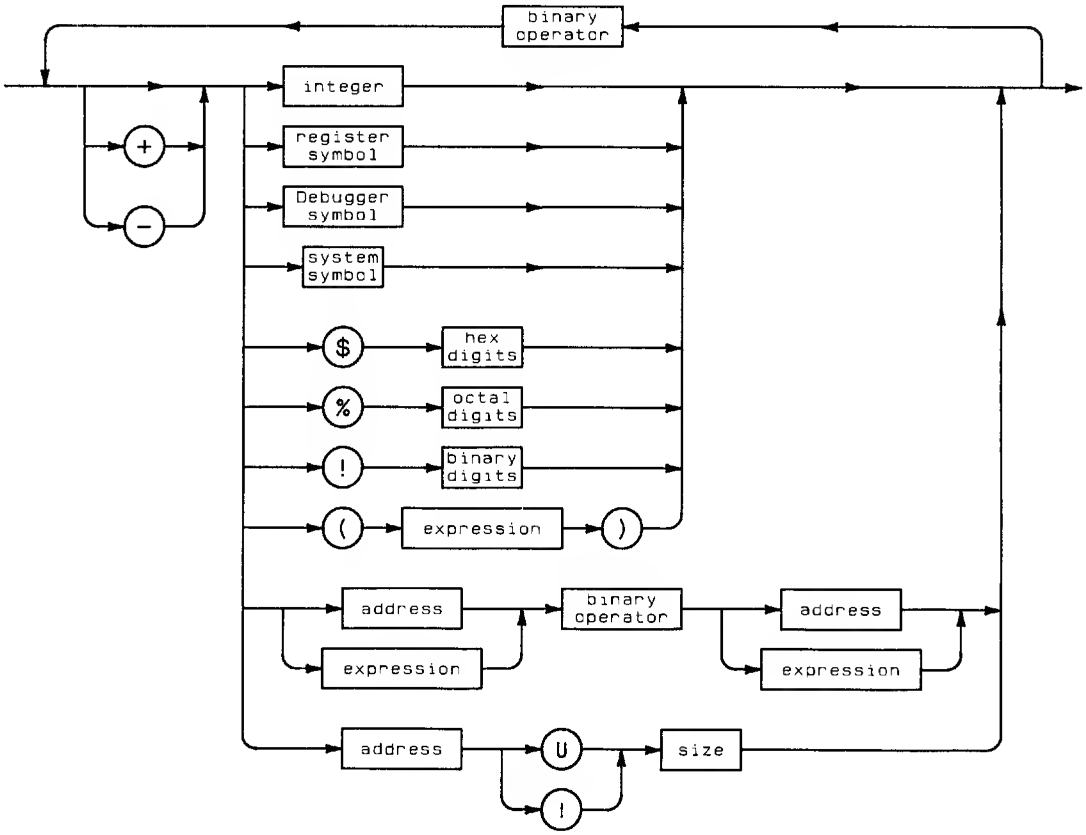
WR – The SF is returned to the current stack frame.

Debugger Command Reference

This section contains a formal description of syntax and semantics for each Debugger command.

Debugger Expressions

With the Debugger, all expressions are *integer* expressions.

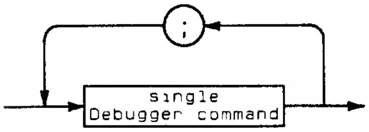


Item	Description/Default	Range Restrictions
binary operator	an operator that requires two operands	+ , -, /, *, <, <=, =, >=, >, <>
register symbol	a symbol representing a processor register	A0..A7, D0..D7, PC, SP, US, SR
Debugger symbol	a symbol known to the Debugger	LN (Line Number) EC (Escape Code) IO (I/O result code) GB (the Global variable Base) RB (the code Relocation Base) SF (the current Stack Frame pointer)
system symbol	any symbol in the system symbol table	–
address	an integer numeric expression followed by a "^", which refers to the contents of the specified memory address	-2^{31} thru $2^{31} - 1$
size	integer expression that specifies the number of bytes to be used	1 thru 4

The "U" (unsigned integer) and "I" (signed integer) option paths indicate whether the value at the specified address and with specified number of bytes (size) is to be treated as a signed or unsigned integer.

Multiple Commands on a Line

Several commands may be entered on the same line. These commands are separated by a semicolon (;).

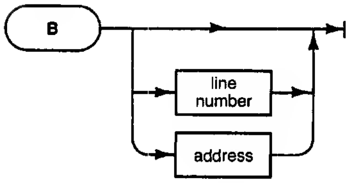


Breakpoint Commands

Breakpoints are points in a program where execution may be halted. The Breakpoint commands control program execution by setting up, activating, and clearing breakpoints in a program.

B

The "B" command causes the breakpoint table to be displayed.

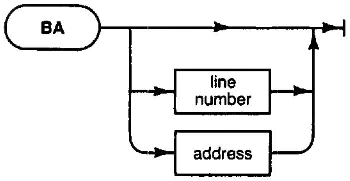


Item	Description/Default	Range Restrictions
line number	an expression that identifies a program line	0 thru 2^{16}-1
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$

The first column contains an "A" for an active breakpoint or a "D" for a deactivated breakpoint. If no location is specified, the table displays all breakpoints.

BA

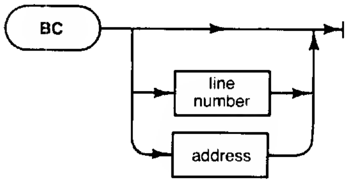
The "BA" command Activates disabled breakpoints. If a location is specified, then only that breakpoint is re-activated; otherwise, all breakpoints are re-activated.



Item	Description/Default	Range Restrictions
line number	an expression that identifies a program line	0 thru 2^{16}-1
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$

BC

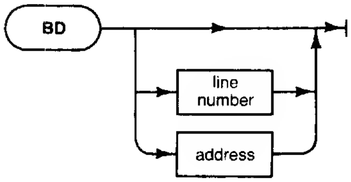
The "BC" command Clears breakpoints. If a location is specified, then only that breakpoint is cleared; otherwise, all breakpoints are cleared.



Item	Description/Default	Range Restrictions
line number	an expression that identifies a program line	0 thru 2^{16}-1
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$

BD

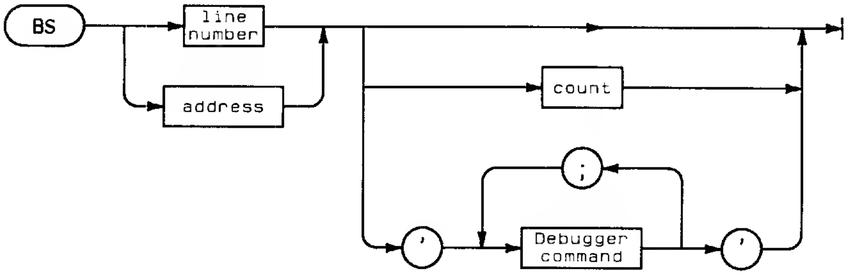
The "BD" command De-activates breakpoints. If a location is specified, then only that breakpoint is de-activated; otherwise, all breakpoints are de-activated.



Item	Description/Default	Range Restrictions
line number	an expression that identifies a program line	0 thru 2^{16}-1
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$

BS

Setting breakpoints with the "BS" command causes the program to stop or perform some operation at a given line number or instruction address.



Item	Description/Default	Range Restrictions
line number	an expression that identifies a program line	0 thru 2^{16}-1
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$
count	expression	-2^{31} thru $2^{31}-1$
Debugger command	command(s) to be executed when breakpoint reached	any legal Debugger command(s)

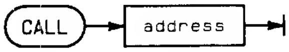
If only a location is specified, the breakpoint is set at that location and then activated. The program will halt just *before* it subsequently reaches that point.

Specifying a count sets a breakpoint that will halt the program after the count has been decremented to 0. (The count is decremented each time the location is reached.) When the program is halted, this type of breakpoint is automatically cleared. (The other two types of breakpoints set with the BS command are not cleared when encountered.)

Adding a command string to the breakpoint causes the command to be executed each time the point is reached. A "?" in the command string causes the Debugger to wait for input from the keyboard. Otherwise, the command is executed and program execution resumes.

The Call Command

This command is used to call the subroutine at the specified address.



Item	Description/Default	Range Restrictions
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$

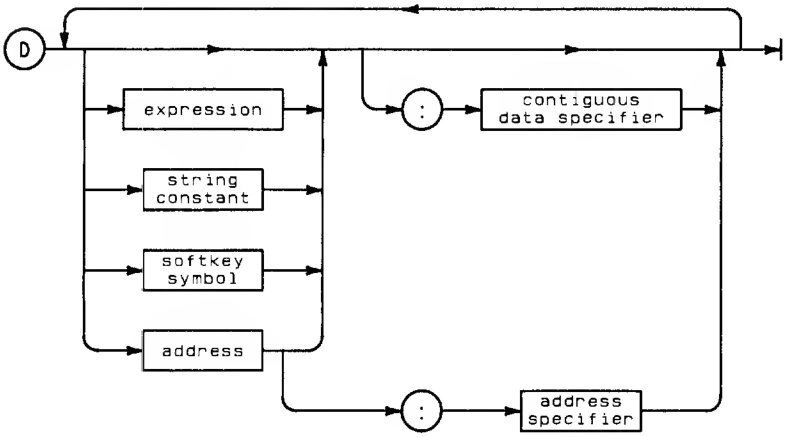
The effect of this command is as if a Jump to Subroutine (JSR) instruction was encountered just before the current program counter (PC).

The CALL command can be abbreviated with the letters CA.

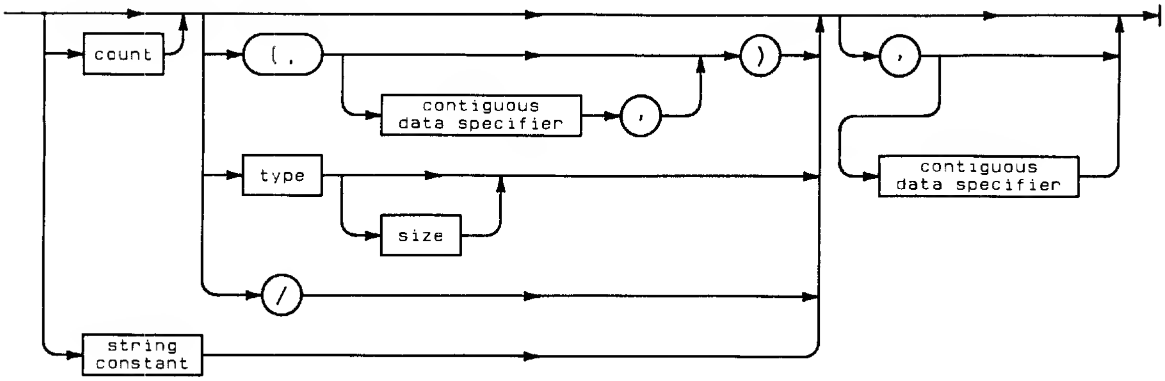
Display Command

D

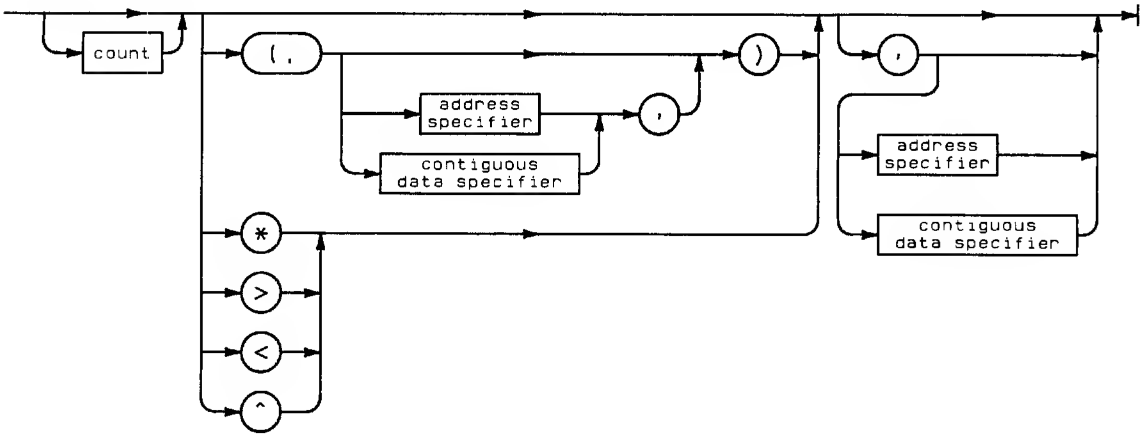
The D command is like a print statement where the parameters are objects and formats.



contiguous cata specifier:



address specifier:



Item	Description/Default	Range Restrictions
expression	integer expression	-2^{31} thru $2^{31}-1$
string constant	literal value	any character delimited with single or double quotes
softkey symbol	a symbol (not the actual key)	K0 thru K9
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$
count	integer constant	1 thru $2^{31}-1$
contiguous data specifier	specifier that identifies the format of data which is contiguous in memory (i.e., memory pointer symbols * etc. not used)	see drawing (nesting limit is 3)
address specifier	specifier that identifies an address in memory (memory pointer symbols such as * may be used)	see drawing (nesting limit is 3)

type	A = Alpha character
	B = Binary
	H = Hexadecimal
	I = Integer (size = 1..4)
	O = Octal (size = 1..4)
	S = String type (size is declared size)
	R = Real (size not allowed)
	U = Unsigned integer
	X = reverse assembly (size not allowed)	

size	integer constant	-2^{31} thru $2^{31}-1$ (except where noted above)

Objects can be immediate, direct, or indirect.

Formats describe the internal representation of the data. Non-consecutive data can be displayed using the format options available when the address parameter is used.

If a format of type S includes a count parameter, then a size parameter must also be included.

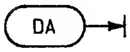
If the output of a Display command fills the screen, a MORE prompt will be issued. A reply of (Return), (Enter), or (Select) ((EXECUTE)) will continue the display. (Shift)-(Select) ((SHIFT)-(EXECUTE)) will cancel the display and the rest of the command string. All other responses will be ignored.

Dump Commands

These commands allow you to perform the DUMP ALPHA and DUMP GRAPHICS functions while in the Debugger.

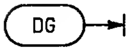
DA

The DA command performs the DUMP ALPHA function.



DG

The DG command performs the DUMP GRAPHICS function.



Note

These commands can only be used while executing programs in the processor's "user mode." If attempted while in "supervisor mode," the following error will be reported:

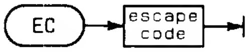
```
NOT ALLOWED NOW
```

Escape Code Commands

These commands allow you generate and trap escape codes while in the Debugger.

EC

The effect of executing this command is the same as if you had executed an ESCAPE(code) in the program just before the current PC. If any ET, ETC, or ETN commands have been used to set up escape code trapping, then the Debugger will be halted and the escape code displayed on the screen.



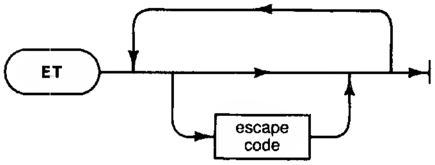
Item	Description/Default	Range Restrictions
escape code	signed integer expression; negative for system escapes, positive for user escapes.	-2^{15} thru 2^{15}

Here is an example display:

```
>EC 10
-EXCEPTION-
ESCAPE CODE          +10
SR=$0004 PC=     -228230 LINE          +9
```

ET

The Escape Trap command allows you to specify that either all escape codes or specified escape codes are to be trapped by the Debugger.



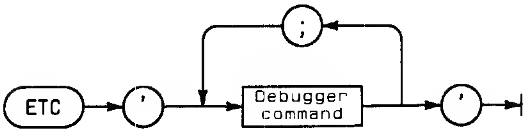
Item	Description/Default	Range Restrictions
escape code	signed integer expression; negative for system escapes, positive for user escapes	-2^{15} thru $2^{15}-1$

If an escape code that is in the list is encountered, execution stops and control is given to the Debugger. If no escape codes are specified, then processing stops for all escape codes.

Up to 4 escape codes may be specified with the ET command.

ETC

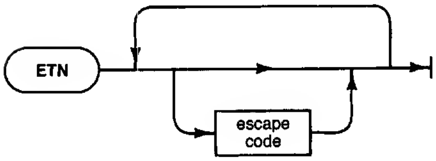
The Escape Trap Command allows you to set up command(s) to be executed when an ESCAPE is generated.



Item	Description/Default	Range Restrictions
Debugger command	command to be executed when an escape is encountered	any valid Debugger command

ETN

The Escape Trap Not command specifies that processing should stop for all excape codes *except* the ones listed. If none are listed, then processing won't stop for *any* escape codes.



Item	Description/Default	Range Restrictions
escape code	signed integer expression; negative for system escapes, positive for user escapes	-2^{15} thru $2^{15}-1$

If the program was started with the D command, then `ETN -20 0` (which traps all except the (Stop) key and normal program termination) is in effect.

Up to 4 escape codes may be specified with the ETN command.

Format Commands

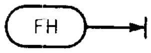
The format commands allow you to specify the default display format.

FB

The Format Binary command sets the default format to Binary values.

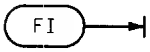
FH

The Format Hex command sets the default format to Hexadecimal values.



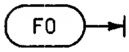
FI

The Format Integer command sets the default format to signed Integer values.



FO

The Format Octal command sets the default format to signed Octal values.



FU

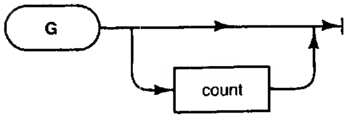
The Format Integer command sets the default format to Unsigned integer values.

Go Commands

The Go commands control program execution by telling the Debugger how many lines to execute or the line at which to halt.

G

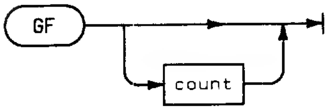
The "G" command causes normal execution to resume. If a count option is used, that number of statements are executed.



Item	Description/Default	Range Restrictions
count	integer expression	1 thru $2^{15}-1$

GF

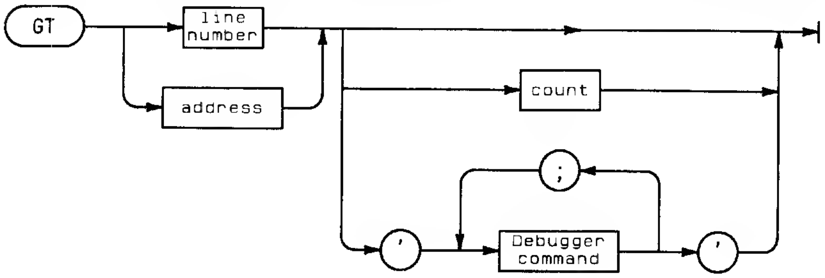
The "GF" command is the same as the "G" command except execution is slowed and line numbers are Flashed in the lower right corner of the CRT.



Item	Description/Default	Range Restrictions
count	integer expression	1 thru $2^{15}-1$

GT

The "GT" command causes execution to Go 'Til the specified location is reached.



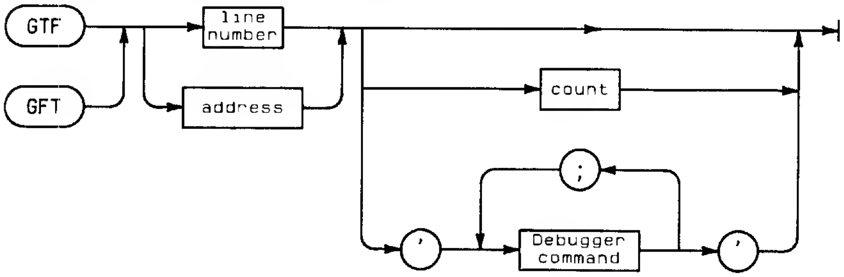
Item	Description/Default	Range Restrictions
line number	an expression that identifies a program line	0 thru $2^{16}-1$
address	integer expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$
count	integer constant	1 thru $2^{31}-1$
Debugger command	command(s) to be executed when the specified line or address is reached	any legal Debugger command(s)

If a count option is used, execution continues until the location is reached that number of times.

If the Debugger command option is used, the command(s) are executed when the location is reached.

GTF

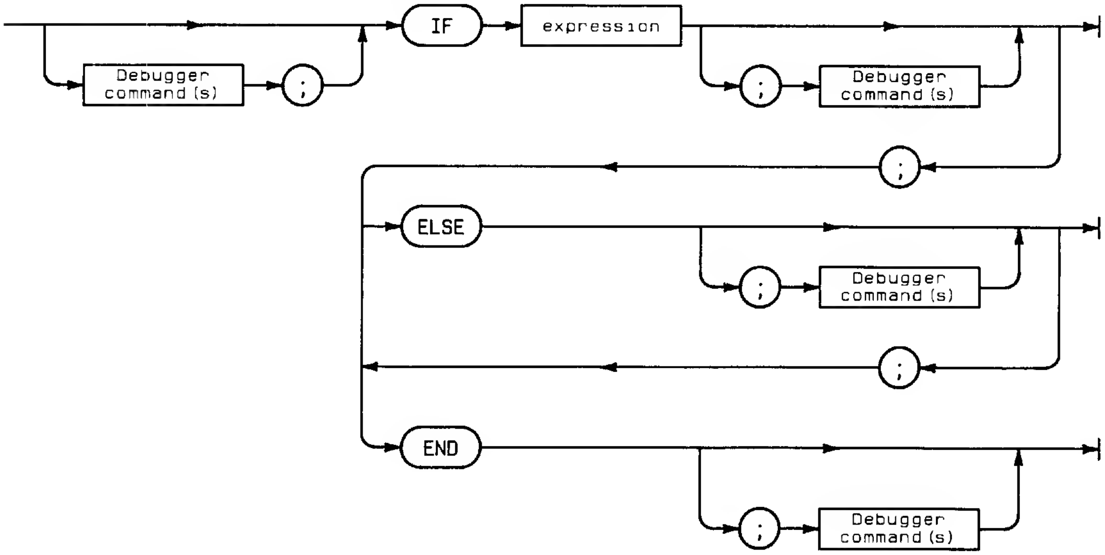
The "GTF" command is the same as the "GT" command except execution is slowed and line numbers are flashed in the lower right corner of the CRT.



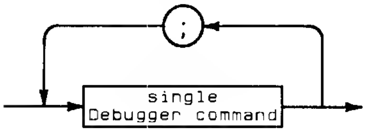
Item	Description/Default	Range Restrictions
line number	an expression that identifies a program line	0 thru $2^{16}-1$
address	integer expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$
count	integer constant	1 thru $2^{31}-1$
Debugger command	any legal Debugger commands delimited with single or double quotes	

IF, ELSE, and END Commands

These commands allow conditional execution of Debugger commands.



Debugger command (s):



Item	Description/Default	Range Restrictions
Debugger command(s)	command(s) to be executed when the sense bit is TRUE	see drawing
expression	numeric or boolean expression whose value determines the state of the "sense bit"	any valid Debugger expression
single Debugger command	one Debugger command	any valid Debugger command described in this reference section

In order to better understand how IF, ELSE, and END statements work, you need some background information. There is a sense bit that determines whether or not Debugger commands are executed. This sense bit is set to TRUE at the beginning of every command line. Commands on the line are executed as long as this bit is TRUE and skipped when the sense bit is FALSE.

When an IF statement is encountered, the expression is evaluated. If it evaluates to non-zero or TRUE, then the sense bit is set TRUE. Subsequent commands are executed while this bit is TRUE. When an ELSE command is encountered, the sense bit is complemented (i.e., if it was TRUE, then it is set to FALSE, and vice versa). When an END statement is encountered, the sense bit is set to TRUE. Here is an example of this situation:

```
>IF 1=1;D 'NON-ZERO';D 'TRUE';ELSE;D 'ZERO';D 'FALSE';END;D 'ALWAYS'
NON-ZERO
TRUE
ALWAYS
```

Here is an example of the converse situation.

```
>IF 0;D 'NON-ZERO';D 'TRUE';ELSE;D 'ZERO';D 'FALSE';END;D 'ALWAYS'
ZERO
FALSE
ALWAYS
```

Notice that the commands after the END statement are always executed. Note that the IF statement does not have to be the first command in the line.

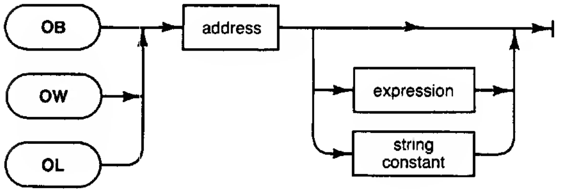
The ELSE command can be abbreviated as EL; the END command can be abbreviated as EN.

Open Memory Commands

These commands allow you to examine, and optionally modify, the contents of memory locations.

OL, OW, OB

The Open Byte, Open Long, and Open Word commands are used to examine consecutive memory locations and to assign values to the locations.



Item	Description/Default	Range Restrictions
address	an expression, followed by a "^", that identifies a location in memory (with OW and OL, the address must be an even number)	-2^{31} thru $2^{31}-1$
expression	integer expression	-2^{31} thru $2^{31}-1$
string constant	literal	any character delimited with single or double quotes

Semantics

When no value is specified after the location, the location and the contents of the location are displayed and followed by a special prompt. The prompt is for an up-arrow key, a down-arrow key, or the (Return) or (Enter) key.

- The up-arrow key causes the next higher location and value to be displayed and the special "Open" prompt.
- The down-arrow key is the same except that the next lower address is displayed.
- The (Return) or (Enter) key causes termination of the "Open" prompt and a return to the standard Debugger prompt.

The amount of the increment/decrement is as follows:

- 1 byte for the "OB" command
- 2 bytes for the "OW" command
- 4 bytes for the "OL" command

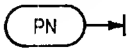
When the Open memory commands are used with value options, the specified value is assigned to the corresponding location. No attempt is made to read the corresponding memory location.

Procedure Commands

These commands allow you to halt the program when a procedure is called or exited. Both of these commands will only work if the procedures were compiled with $DEBUG ON$.

PN

The PN (Procedure Next) command halts the Debugger when a procedure is called by the current procedure or main program (or when the current procedure is exited).

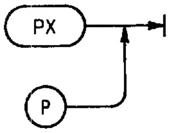


When the current procedure or main program calls another procedure, the Debugger displays `NEXT PROC` and halts the program *before* executing the first line of the called procedure.

If the current procedure is exited before another is called, the message `PROC EXITED` is displayed and the Debugger is halted *before* executing the first line of the procedure that called the current one.

PX

The PX (Procedure eXit) command allows you to halt program execution when the current procedure is exited.



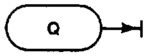
When the current procedure is exited, the message `PROC EXITED` is displayed and the program is halted *before* executing the next line of the procedure that called the current one. Calling a procedure while in the current one is not reported (as it is with PN).

Queue Commands

The Queue commands control and display the Queue, which is a record of the line numbers of statements (or memory addresses of processor instructions) encountered during the execution of a program.

Q

The "Q" command displays the addresses or line numbers and addresses of the most recent statements executed since a "QS" command or the start of execution of the current program.



"MORE" is given as a prompt when part of the Queue has been displayed and there is more to come; a reply of (Return), (Enter), or (Select) ((EXECUTE)) will cause the next 1..21 Queue entries to be displayed. Any other reply will be interpreted as another command.

QE

"QE" ends the recording of information in the Queue

QS

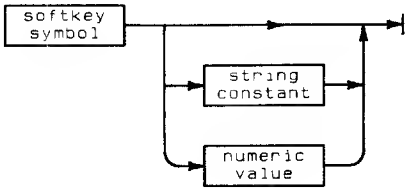
"QS" starts the recording of information in the Queue

Softkey Commands

The Softkey commands allow you to define System softkeys to display literal values and commands, so that these keys will be used as typing aids.

"K0" thru "K9"

These commands allow you to define the softkeys as typing-aid keys; when the softkey is subsequently pressed, it puts the string constant or the result of evaluating the integer expression into the Debugger command-input line.



Item	Description/Default	Range Restrictions
softkey symbol	literal symbol that denotes a softkey	K0 thru K9
string constant	literal value	any characters delimited with single or double quotes
numeric value	integer expression	-2^{31} thru $2^{31}-1$

Numeric values or command strings can be assigned to the softkey symbols K0 through K9 by typing the key *symbol* (not by pressing the actual key) and then typing the value to be assigned to the key.

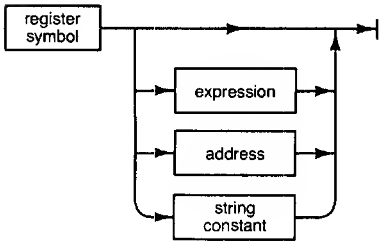
If a string constant is assigned to the softkey symbol, subsequently pressing the corresponding softkey will cause the literal value to be placed in the Debugger command-input line. If a numeric expression is assigned to the softkey symbol, the *result* of evaluating the expression is placed in the input line.

After the command on a line is completed, pressing (Return), (ENTER), or (Select) ((EXECUTE)) causes the line to be interpreted.

To see the string constant or numeric value that is currently assigned to a softkey symbol, type the *symbol* and press (Return).

Register Operations
A0...A7, D0...D7, SP, US, SR, PC

Registers can have their contents displayed or altered. If a value follows the register symbol, that value is assigned to the register. Otherwise, the current value of the symbol is displayed. Without the assignment, the command is the same as the D command.



Items	Description/Default	Range Restrictions

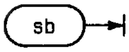
register symbol	A0...A7, D0...D7, SP, US, SR, PC
	AA = All Address registers
	DD = All Data registers	

expression	integer numeric expression	-2^{31} thru $2^{31}-1$
string constant	any characters delimited with single or double quotes	

"AA" and "DD" cannot be used to assign values.

System Boot

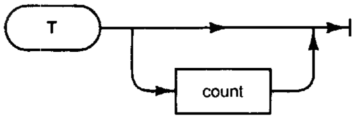
The "sb" command places the computer in the "power-up" state for rebooting. The command must be typed in lowercase.



Trace Commands

T

The "T" command with count specification causes that number of machine instructions to be executed. A "TD" command is executed after each machine instruction.

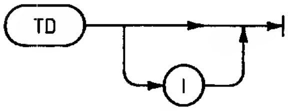


Items	Description/Default	Range Restrictions
count	integer expression	1 thru $2^{31} - 1$

For use with Assembly Language debugging. This command ignores address break points.

TD

The TD (Trace Dump) command displays the command string which is defined by the f4 (k4) softkey.

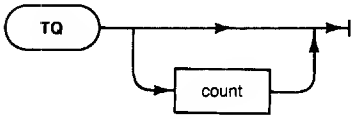


At power up, f4 is defined to display the PC, the instruction at (PC), the status register, the SP, and all the A (address) and D (data) registers. This display may be altered by changing the definition of f4.

The optional parameter "I" restores the initial definition to softkey f4.

TQ

TQ (Trace Quiet) is the same as the T command except a TD command is only executed after the last instruction.

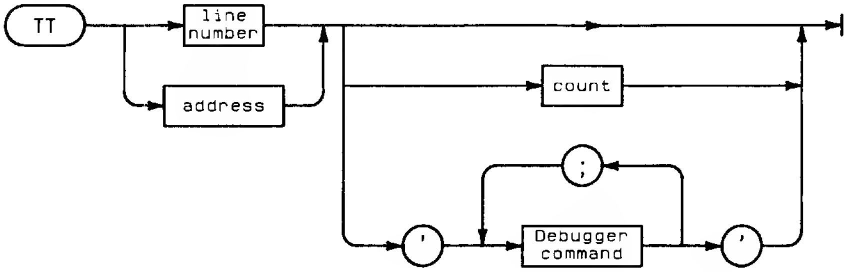


Items	Description/Default	Range Restrictions
count	Integer expression	1 thru $2^{31} - 1$

For use with Assembly Language debugging. This command ignores address break points.

TT

TT (Trace 'Til) causes machine instructions to be executed until the specified line number or address is reached. TT is like TQ in that a TD command is executed only after the last statement or instruction is executed.



Items	Description/Default	Range Restrictions
line number	an integer expression identifying a program line.	0 thru $2^{16}-1$
address	an integer numeric expression followed by a "^"	-2^{31} thru $^{31}-1$
count	integer expression	-1 thru $2^{31}-1$
debugger command	any legal commands delimited with single or double quotes	

For use with Assembly Language debugging. This command ignores address break points.

Walk The Procedure Links

WD

Walk the Dynamic link. This command causes the stack frame pointer to move to the stack frame of the calling procedure.

WR

Walk Reset. Brings the stack frame pointer to the stack frame of the current procedure.

WS

Walk the Static link. This command is the same as the "WD" command except that the stack frame pointer takes the value of the static link. This brings you to the stack frame of the nesting procedure as opposed to the calling procedure. Level 1 procedures have no static link.

Special Configurations

Chapter
9

Introduction

HP Series 200 Workstation Pascal is self-configuring. As it boots, interface/device driver modules in the Initialization Library (BOOT:INITLIB) are loaded into memory and initialized. Then, the TABLE program determines what peripheral devices are connected to the computer (such as local and remote mass storage devices, printers, and so forth); if the driver module(s) for a particular interface or device are in memory, then the TABLE program can usually assign to it a logical unit number which makes it accessible to the File System.

The term "standard configuration" is defined to be any combination of computer and peripheral devices that will be configured by the Pascal 3.0 system as it is shipped. This chapter describes how to change this "standard" system configuration.

Chapter Organization

This chapter contains many sections; however, they can be essentially split into three categories.

- A description of how the system boots and auto-configures itself.
- Brief descriptions of several possible configurations.
- Procedures for making changes to the "standard" configuration.

You will probably want to read about how the system boots and auto-configures itself, regardless of whether you want to change your system's configuration. Next, you will probably want to scan the possible "non-standard" ways that you can configure your system. The following sections briefly describe several common configurations:

- Changing hard-disc volume sizes
- Setting up several bootable system configurations
- Adding interfaces and peripheral devices
- Setting up the SRM System
- Changing system printers
- Adding Bubble and EPROM cards
- Using alternate directory access methods (DAMs)

Then, when you know which configuration change(s) you want and which of the procedures you will need to use to make the changes, you can follow the procedures in the third major section of the chapter. These procedures are as follows:

- Coalescing logical volumes on hard discs into larger volumes
- Copying system files and changing their names
- Making an AUTOSTART or AUTOKEYS stream file

- Adding driver modules to INITLIB
- Modifying the auto-configuration program (CTABLE)
- An example SRM configuration

As an example of the first category, suppose you want to connect one HP 9133V Hard Disc Drive and one HP 7912 Disc Drive to your workstation. The standard TABLE program assumes that it should assign 4 unit numbers to the 9133V hard-disc drive and 30 to the 7912. However, since it reserves only 30 unit numbers for *all* hard-disc volumes, the standard TABLE will not be able to access all 34 volumes (if that is the way that these discs have been or will be partitioned); it will either recognize all 4 volumes on the 9133V and only the first 26 on the 7912, or all 30 on the 7912 and none on the 9133V. Probably the easiest way to make all parts of both discs accessible is as follows: first, "coalesce" the *last* 5 logical volumes on the 7912 into one larger volume (to change the number of logical volumes to 26); second, set up the hardware so that the 9133V gets the lower unit numbers (11-14) and the 7912 gets higher numbers (15-40). The alternative way to make all parts of both discs accessible is to modify the the standard TABLE program; the source program is called CTABLE.TEXT, and it is supplied to you on the CONFIG: disc.

An example of the second category was given in the *Pascal 3.0 User's Guide*. The system files (such as EDITOR, FILER, and so forth) were copied to a hard disc (913x family). No file names were changed. An example AUTOSTART file (the third category) was given in the same guide. It P-loaded some system files.

As an example of the fourth category, suppose you want to use an HP 98625 High-Speed Disc interface and an HP 98620 DMA Controller card with a CS80 disc drive. First, add module DISC_INTF to the INITLIB file (modules DMA and CS80 are in the INITLIB file supplied with your system). Then when the system is subsequently booted, the standard TABLE program will, barring other restrictions, automatically recognize the disc and make it accessible. (An alternate but less "permanent" way would be to eXecute module DISC_INTF after booting the system and then eXecute TABLE again.) You will then probably want to copy most system files to the CS80 disc, which is another example of the first category.

As an example of the fifth category, suppose you want to connect two HP 7912 Disc Drives to your workstation. The standard TABLE program will not make both drives accessible, since it assumes that each disc needs to be allocated 30 unit numbers and assigns all 30 units available for hard discs to the 7912 with the highest priority. In order to access both drives with the File System, you will need to modify the standard TABLE program ("coalescing" will not work in this case). In this type of situation, you may want to change the default number of logical volumes that the system creates on each drive. After re-compiling and then running the properly modified program, the system will recognize and allow you to access all parts of each drive. You will probably want to replace the original TABLE program with the new version so that this configuration will automatically be made at the next power-up and system boot time.

The Booting Process

This section explains what is going on within the machine as the Pascal System is loaded. It is intended to give you a few more insights into how the system works. It does not, however, describe *how* to boot your Pascal system; that topic is covered in the *Pascal 3.0 User's Guide* provided with your computer. Neither does it describe modules; that topic is covered in the Compiler chapter.

The Boot ROM

Inside the computer is a ROM (Read-Only Memory) that contains the information needed to begin loading an operating system. The loading process is often called "booting" because it is the computer's way of "pulling itself up by its own bootstraps." This ROM is therefore called the "Boot ROM". The Boot ROM is a non-volatile storage device; its contents are not lost when power is removed.

There are currently several different versions of the boot ROM. Thus, the booting process is slightly different depending on which version of boot ROM is in your computer. However, all perform the general steps outlined in this section.

When you power-up, the computer's central processing unit (CPU, which is a 68000-family processor) reads the first few bytes of this ROM, which begins at address 0. These bytes contain such information as the address of the first executable machine-language instruction and initial value of the stack pointer. After loading these values, the processor continues executing routines in the Boot ROM.

The processor next executes routines that perform a self-test and then displays the amount of memory installed in the computer. You may not see the amount of memory displayed if the CRT is just warming up. After self-test, the processor executes another Boot ROM routine that searches various mass storage devices (such as disc drives) for an operating system; the Boot ROM recognizes files of type "Systm" and with name beginning with the letters "SYSTEM_" (or "SYS" with Boot ROM 3.0 and later) as being operating systems. It also searches system ROM for ROM-based systems (such as BASIC).

Depending on its version and how many systems it finds, the Boot ROM will either choose a system or let you choose one (for instance, Boot ROM 3.0 and later versions allow you to choose one if you intervene in the boot process). With Pascal, this "Systm" type file is called "SYSTEM_P" and it will be discussed momentarily.

The Pascal System Discs

The Pascal system is delivered on either 5.25-inch (mini-floppy) or 3.5-inch (micro-floppy) flexible discs. The discs contain the operating system, subsystems like the Editor and Compiler, and several libraries and utility programs. The discs that you received are listed in the *Pascal User's Guide*. You will have to boot Pascal from these discs at each power-up unless you reconfigure your system. The disc called BOOT: contains the SYSTEM_P file that will be loaded into memory first.

The System Boot File (SYSTEM_P)

The BOOT:SYSTEM_P program is an absolute-addressed program that contains the bare minimum Pascal operating system "kernel." (It was created using the Librarian's Boot command.) It is absolute-addressed so that the Boot ROM can use a simple loading routine.

The SYSTEM_P file consists of a linking loader (more elaborate than the loader found in the Boot ROM) and a few support routines. This kernel is loaded into volatile read/write memory (also called random-access memory, or RAM) from non-volatile memory (usually discs). The linking loader then loads the rest of the system.

In this system, there is no "kernel" in the closed sense of the term, such as a closed system like HP-UX. The system has an open design which allows modules to be added to the system – while the system is running. However, the term "kernel" will still be used in this text to describe the minimum working environment.

The loader then continues by completing construction of the operating system by loading the "Initialization Library" called INITLIB, which is also on the BOOT: disc.

The Initialization Library (INITLIB)

This BOOT: library file consists of modules that complete the kernel of the Pascal operating system. (Some of the modules are programs.) These modules mainly provide access methods (or device "drivers") for internal interfaces and peripheral devices.

Installing INITLIB Modules

As each INITLIB module is loaded into memory, it is bound to the operating system by a linking process. After the loading is complete, each program is executed once. The programs in INITLIB are referred to as "installation code;" their purpose is to properly initialize variables or allocate storage that will be used by these modules. Many interface-driver modules check to see if the interface they are to drive is there, and if not they don't install themselves.

Once INITLIB is loaded and the installation code has been executed, the system has found and identified all interface cards installed in the machine; however, no scan has been made for peripheral devices.

Auto-configuration of a peripheral device requires the device's driver(s) to be in memory at the time that the TABLE program is run (TABLE will be discussed in the next section). The HPIB module is an example of a driver for HP-IB interfaces. If the driver is part of the INITLIB file, then the device can be interrogated at a later time by TABLE (unless other conditions restrict). If the driver is not in INITLIB, then you must add it to the file (or alternately load the driver into memory by executing the installation program that contains the driver module).

Adding and Removing INITLIB Modules

Since the operating system is an "open kernel," you can add, replace, or delete modules within this library; more details regarding these operations are described in the Adding Modules to INITLIB section of this chapter. You must not change the order of modules in this library; neither should you link them together (with the Librarian), which would result in rendering the programs non-executable.

Module LAST

The last piece of installation code in INITLIB is the program named LAST, which attempts to execute the BOOT: files named STARTUP and TABLE. Here is the algorithm used to load and execute these two files; each file's function is described in a subsequent section.

1. If STARTUP is found on the "Boot volume" (i.e., the same volume on which the Systm file, such as SYSTEM_P, was found), then that program is loaded (but not executed).
2. LAST then looks for TABLE on the Boot volume. If TABLE is found there, then it is loaded and executed; it makes File System volumes accessible. If TABLE is not found, then only the keyboard, screen, and Boot volume will be accessible (see the brief description in the subsequent section called Failure of the TABLE Program).
3. If STARTUP was not found on the Boot volume, then the Boot ROM looks for it on the current system volume (at this point it might not be the Boot volume, because TABLE may have re-defined it).
4. STARTUP is then executed.

The Command Interpreter (STARTUP)

With the Pascal system delivered to you, the BOOT:STARTUP file is the Command Interpreter (or Main Command Level) program. However, you can write any program, optionally Link it with the Librarian, name it STARTUP, and with it replace the existing STARTUP file. It will then be loaded at power-up, instead of the Command Interpreter program.

If you use your own STARTUP program, be careful not to destroy the original STARTUP program. The recommended method is to use the Filer's Filecopy command to make a copy of the BOOT: disc on a blank initialized disc and then replace the STARTUP program with your new STARTUP program on that disc. Use the disc with the new STARTUP to boot the computer and your program will start running instead of the Pascal operating system.

The Auto-Configuration Program (TABLE)

The purpose of the TABLE program is to make devices accessible to the File System. Since this is one of the principle topics of this chapter, the subsequent section called Auto-Configuration is devoted to the intricate details of how this program works. For now, let's assume that it has already chosen the system volume and finish this overview of how the system boots.

The AUTOSTART and AUTOKEYS Stream Files

If present on the system volume and if data can be written on the volume (i.e., it is not a read-only volume), the AUTOSTART file is automatically streamed by the system at power-up; if the volume does not permit write operations (such as EPROM cards), then the AUTOKEYS file is streamed, if present. These files must be "stream" files, are sequences of characters that are used by the system just as if they were commands typed from the keyboard (a "command stream"). Stream files are formally described in the Main Command Level Reference section of the Overview Chapter.

The AUTOSTART or AUTOKEYS file must be located on the volume designated as the system volume at the point that the TABLE program has finished execution. There is an AUTOSTART file on the BOOT: disc. Here are the contents of the AUTOSTART file provided with your system.

```
1MARO

xSWVOL
SYSVOL
3
wsSYSVOL:
qv
```

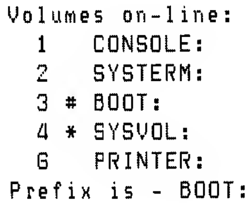
If you use the original BOOT: disc on a single drive system, it is the AUTOSTART file which causes you to be instructed to place SYSVOL: in the drive and then press the (X) key. This AUTOSTART file then changes the system volume to "SYSVOL:". But because BOOT: is initially the system volume, the AUTOSTART file is found and executed. On dual-drive systems, the media in the second disc (nominally SYSVOL:) will usually become the system volume.

Libraries

The Pascal system is shipped with many library modules. Some are device drivers (in BOOT:INITLIB or on the CONFIG: disc), while others provide procedures, etc. for applications such as device I/O and graphics (on the SYSVOL:, LIB:, and FLTLIB: discs). Once the system has booted successfully, you can use these libraries. INITLIB modules are described in the Adding Modules to INITLIB section of this chapter. Application libraries are fully discussed in the *Pascal 3.0 Procedure Library* and *Pascal 3.0 Graphics Techniques* manuals. You can also write your own libraries, as described in the description of Modules in the Compiler chapter.

The Auto-Configuration Process

A device is only accessible to the File System if it has been assigned a logical unit number. You may be familiar with the Filer's Volumes command, which shows the correspondence between logical unit numbers and volumes. Here is a typical display:



```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 # BOOT:
  4 * SYSVOL:
  6   PRINTER:
Prefix is - BOOT:
```

The Unit Table

To make devices accessible to the File System, TABLE fills in entries of the Unit Table so as to correctly associate logical unit numbers with logical volumes (and the software required to access the devices on which those volumes exist). The Unit Table is actually a global system pointer variable called "Unitable," which points to a table that contains 50 entries – one for each logical unit (and potential volume). The Unit Table variable is accessed by many parts of the system, such as the Filer, Editor, and Compiler, when they want to use one of the devices.

This section describes how the standard TABLE program assigns Unit Table entries. To see the exact algorithms implemented in Pascal code, refer to the TABLE program called CTABLE.TEXT and corresponding commentary later in this chapter.

Standard Auto-Configuration

The results of a typical auto-configuration process performed by the standard TABLE program are shown in the following table. Each entry is further discussed in subsequent text:

Standard Unit Table

Unit	Nominal Assignment
1	System CRT Screen (CONSOLE:)
2	System Keyboard (SYSTERM:)
3	1st priority floppy (drive 0, primary DAM)
4	1st priority floppy (drive 1, primary DAM)
5	Shared Resource Manager (remote mass storage)
6	System Printer (PRINTER:)
7	2nd priority floppy (drive 0, primary DAM)
8	2nd priority floppy (drive 1, primary DAM)
9	3rd priority floppy (drive 0, primary DAM)
10	3rd priority floppy (drive 1, primary DAM)
11-40	Hard discs (highest to lowest priority)
41	1st priority cartridge tape (LIF DAM)
42	2nd priority cartridge tape (LIF DAM)
43,44	1st priority floppy (same volume as 3 & 4, but alternate DAM)
45	SRM system volume, if appropriate
47,48	2nd priority floppy (alternate DAM for 7 and 8)
49,50	3rd priority floppy (alternate DAM for 9 and 10)

How Unit Numbers Are Assigned

In the Unit Table, certain unit numbers are preferentially assigned to particular classes of devices. Here are the general classes of devices:

- Unblocked devices (i.e., "byte stream" devices that do not have directories) like the keyboard, screen, and local printers
- Floppy disc drives (including 5.25-inch, 3.5-inch, and 8-inch)
- Hard disc drives
- SRM systems
- DC600 cartridge tape drives

The floppy and hard disc drives and tape drives are all "blocked" devices.

Unblocked Devices

To fill the Unit Table, the TABLE program assumes that "unblocked" devices, such as the screen (CONSOLE:), the keyboard (SYSTERM:), and system printer (PRINTER:) are always present and assigns them to units #1, #2, and #6, respectively. However, it must scan for the presence of "blocked" devices (i.e., mass storage devices with directories). Once these devices are found, their locations (select code, HP-IB address, etc.) and attributes (type of disc drive, capacity, etc.) are put in the table entry corresponding to the logical unit number.

Blocked Devices

Here are the steps that the standard TABLE program goes through in assigning unit numbers to blocked devices.

Interfaces and Devices Scanned

In order to find mass storage (blocked) devices, the TABLE program first scans interface select codes 7, 8, and 14 for the presence of an HP-IB type interface: select code 7 is the built-in HP-IB interface; select code 8 is the factory default setting for optional HP-IB interfaces; 14 is the factory default setting for HP 98625 High-Speed Disc interface (a fast HP-IB interface).

If an HP-IB interface is found, addresses 0 thru 7 are interrogated for the presence of blocked devices. (Most HP-IB peripherals identify themselves when asked politely.) The purpose of this interrogation is to determine what type of device (such as what family of disc drive, capacity of drive, etc.) is present at each location.

Device Classes and Unit Numbers

The TABLE program makes a list of the devices found in each of these classes:

- Floppy discs – this class includes all 5.25-inch, 3.5-inch, and 8-inch floppy disc drives, and all CS80 or SS80 devices that have a single *physical* volume with capacity less than 10 Megabytes.
- Hard discs – this class includes all 913x hard discs, and all CS80 or SS80 devices that have either multiple *physical* volumes or a single *physical* volume with capacity greater than or equal to 10 Megabytes.
- Tape drives – this class includes all DC600 cartridge tape drives, such as the HP 9144 Tape Drive as well as tape drives integrated into the CS80 Disc/Tape Drives.

Up to 10 devices can be on the list for each class. If more than 10 devices are found in a class, then only the *last* 10 found are maintained in the list.

As shown in the preceding Standard Unit Table diagram, groups of unit numbers have been reserved for each particular class of devices. For instance, unit numbers 3 and 4, and 7 through 10 are reserved for floppy discs. Unit numbers 11 through 40 are reserved for hard discs. Unit numbers 41 and 42 are reserved for tape drives.

Device Priority

The "priority" of a device is generally as follows: the later in the scanning sequence a device is found, the higher its priority is. Remember that interfaces are scanned in the order of select codes 7, 8 and 14; and on each HP-IB interface, addresses 0 through 7 are interrogated. Thus, a device at 702 has higher priority than a device at 700 but lower priority than one at 800. However, if a device was used to boot the system, then that device will have the highest priority in its category.

Assigning Unit Numbers to Floppy Disc Drives

Units are assigned to floppy discs in pairs, according to device priority. For instance, if two dual-drive floppies are found, then the highest priority floppy device will be assigned unit numbers 3 and 4 and the lower priority device assigned unit numbers 7 and 8. However, if two single-drive floppy devices are found, then the highest priority device will be assigned unit 3 and the lower priority device assigned unit number 7. Up to three floppy drives (and thus pairs of floppy volumes) can be assigned unit numbers.

Assigning Unit Numbers to Hard Disc Volumes

Hard discs are also assigned unit numbers according to device priority; however, there is also another consideration. Since all hard discs currently supported by this system have capacities of several millions of bytes, the standard TABLE prefers to "partition" the *physical* volumes into smaller *logical* volumes. (Some hard discs are also organized to be accessed as four separate physical volumes, rather than one large physical volume; see the subsequent Volume Sizes table for further information).

TABLE sets up Unit Table entries for hard discs according to two factors: the priority of each device, and the number of logical volumes it is assumed to have. Here are the number of units that the standard TABLE will reserve for each hard disc drive, and the corresponding size of each volume.

Standard Hard Discs Volume Sizes

Product Number	Number of Volumes	Volume Size (in bytes)	Volume Size (in sectors)
913x[1] A and V (**Not** Option 10)[2]	4	1 152 000	4 500 (all are same size)
913x A or V (Option 010)[3]	4	1 206 272	4 712 (all are same size)
913x B	9	1 071 360	4 185 (except last = 4340)
913x XV	14	1 031 680	4 030 (except last = 4340)
7908	16	1 030 400	4 025 (except last = 4375)
7911	27	1 032 192	4 032 (except last = 4992)
7912	30	2 179 072	8 512 (except last = 9408)
7914	30	4 390 912	17 152 (except last = 18688)
7933 and 7935	30	13 471 744	52 624 (except last = 53820)

The logical partitioning of hard discs is made by the standard TABLE with the following algorithm. For each device on the list (of up to 10 devices), it calculates the number of volumes required by the device, assuming that the disc is now or will be partitioned; the default number of logical volumes assumed to be on each disc and the size of each volume are shown above. It then begins assigning unit numbers according to device priority; each device is assigned unit numbers according to the number of logical volumes *assumed* to be on the device, regardless of the number of volumes actually on that device. TABLE begins with 11, and continues either until all volumes have been assigned numbers or unit number 40 is reached, whichever occurs first.

At the point that it assigns unit numbers to a device, TABLE has *not* yet determined whether the disc has actually been partitioned. In fact, the disc may not have been initialized yet, or it may have been initialized but not partitioned as assumed. In the second stage of the assignment algorithm, TABLE looks on the disc for each volume's directory. Since these are logical volumes, each directory is assumed to be at an "offset" from the beginning of the disc.

If a valid directory is found at the expected location on the disc (i.e., at the assumed offset), then the corresponding unit number is assigned to the volume. For instance, if a valid directory is found in the first location, then it is assigned the first unit number for that disc (e.g., unit #11 will be assigned to the first directory on the highest priority hard disc device). As each subsequent directory is found, it is assigned the corresponding unit number. For example, if the only hard disc in a system is an HP 9133XV Hard Disc which has been partitioned and initialized according to the standard TABLE volume sizes for this disc, then it will be assigned 14 unit numbers (11-24).

[1] The "x" used here signifies either 9133, 9134, or 9135 products.

[2] The 913x A and V drives (**Not** Option 10) look like 4 separate devices, because they are accessed as *4 separate* "disc units" or "drive numbers."

[3] The 913x A and V Option 10 drives are like the B and XV suffix drives; they are accessed as *1 single* "disc unit" or "drive number."

If a subsequent directory is not found at its expected offset, then that area of the disc is assumed to be part of the last valid directory that preceded this one. For instance, if valid directories are found only in the 1st and 4th expected directory locations on an HP 9133V Hard Disc (assumed to have 4 volumes), then the first volume is assumed to be a coalition of the first three volumes (of the default size) on the disc.

If the first directory is the only valid one found, then the disc is assumed to be one single logical volume. For instance, if the only hard disc in a system is an HP 7911 Hard Disc which has been initialized and partitioned according to the standard TABLE volume sizes for this disc, then it will be assigned 27 unit numbers (11-37). However, if the disc was initialized by the Series 200 BASIC system (or coalesced into one volume using the procedure shown later in this chapter), then it will appear as one single, large volume and assigned only unit number 11. In this case, the last 26 unit numbers allocated for the device (12-37) are *not* usable. If another hard disc (with lower priority) were added to this hypothetical system, then it would be assigned unit numbers beginning with 38, not 12.

Note

The only place that this logical partitioning information is kept is in the Unit Table entries for each volume; however, the information in the Unit Table is used by other parts of the system, such as the MEDIAINIT program that initializes (formats) the disc. The disc drive itself has *no* knowledge whatsoever of this partitioning scheme.

As another example of device priorities, suppose that you had an HP 9133XV drive and an HP 7908 drive in your system. Suppose also that the 9133 is at 702 and the 7908 is at 700. The 9133 is at the higher bus address (and is therefore found after the 7908 is found during the scan sequence), so it has the higher priority (assuming that the 7908 was not the boot device). The standard TABLE presumes that the 9133 is partitioned into 14 logical volumes, so it allocates 14 unit numbers (11-24) for the device. It then allocates 16 unit numbers (25-40) for the 7908 for analogous reasons.

As you might guess from the preceding discussion, even though there may be up to 10 devices in the list of hard discs, not all of the volumes they contain will necessarily be assigned unit numbers and thereby made accessible. Only the volumes to which unit numbers are assigned will be accessible. For instance, if the preceding example would have been two 7908 drives, then the highest priority device will be assigned 16 unit numbers (11-26), while the lower priority drive will only be assigned 14 unit numbers (27-40). If this disc had actually been partitioned into 16 logical volumes, then its last 2 volumes would *not* be accessible.

Note

If you plan to use your hard disc with BASIC, you should set up the disc as one logical volume. See the File Interchange Between Pascal and BASIC section of the Technical Reference appendix.

Choosing the System Volume

The final step made by the standard TABLE is to choose the system volume. The operating system makes use of this volume for several purposes. For instance, after the system volume is designated at boot time, it is then inspected for system files (such as the EDITOR, FILER, and COMPILER). It is where the autostart file (AUTOSTART or AUTOKEYS) is assumed to be. It is also used by the system for storing temporary files that it creates for processes such as expanding Stream files. The system volume should remain on-line at all times if possible.

Here is the algorithm used by the standard TABLE program to determine which volume will be designated as the system volume; the "Boot volume" is the volume from which the BOOT: files named SYSTEM_P, INITLIB, and TABLE were loaded:

1. If a Boot volume was assigned a unit number *and* its capacity is greater than 300 Kbytes, then this device is designated as the system volume.
2. If step 1 did not designate a system volume, then search all volumes in the sysunit_list (a structure declared in TABLE). If a logical volume with a valid directory is found during this search, then it will be designated as the system volume.
3. If neither step 1 or 2 designated a system volume, then use the Boot volume as the system volume.

Failure of the TABLE Program

By the way, if the TABLE auto-configuration program ever fails during the boot process, unit number 6 (normally the standard PRINTER: volume) is assigned to the screen, and unit number 3 is assigned to the "Boot device." You can only execute programs off of unit number 3 (with the Main Level eXecute command); it is otherwise inaccessible to the File System.

If TABLE is executed again but fails during this subsequent execution, then the Unit Table reverts back to its state before this unsuccessful try was attempted. (This is true *every* time that TABLE is subsequently executed.)

Example Special Configurations

This section describes several common types of special configurations. It outlines the general procedures required to implement them. The subsequent section called Modifying the Configuration provides the details of the procedures.

Hard Disc Partitioning

The way that the standard TABLE program prefers to partition hard discs was explained in the preceding discussion of Auto-Configuration. This section briefly describes the two methods of changing this default partitioning.

- "Coalesce" adjacent directories into one larger volume
- Modify the CTABLE program to partition the discs differently

The procedure for coalescing logical hard disc volume is given in the subsequent Modifying the Configuration section.

Coalescing adjacent volumes will work well under these *general* circumstances:

- The total number of volumes that TABLE assumes it will find on *all* discs is less than 30.
- The sizes of volumes that you can make by coalescing an integral number of logical volumes is acceptable (i.e., the "resolution" of the default volume sizes is good enough).

If the desired configuration cannot be made by merely coalescing volumes, then you will have to modify the standard TABLE program (CTABLE.TEXT source file). Modifying the standard TABLE is also described in Modifying the Configuration.

Multiple On-Line Systems

If requested by operator intervention at power-up, computers equipped with Boot ROM 3.0 and later versions find all the on-line system Boot files (for example, SYSTEM_P) and display their names. You can choose the one you want to be booted. For instance, if you have a Pascal and a BASIC system on-line, you can choose which you want to boot.

> **Note**
>
> The term "system Boot file" is used to identify a file that is found and loaded by the Boot ROM, such as SYSTEM_P; this file then loads the corresponding operating system.
>
> The term "BOOT: file" is used to identify a file used during the boot process; these files are on the BOOT: disc shipped with your system.

By modifying certain BOOT: files (usually INITLIB and TABLE only) and uniquely re-naming each different set, you can give yourself the option of choosing different Pascal system configurations at power-up. (This is not possible with the earlier Boot ROMs.)

For instance, suppose you want to have one system version that sets up SRM as the default volume and another that does not allow access of the SRM system. In such a case, you can create two systems, each of which is tuned for the desired usage; you can choose the one you want at power-up. To configure your system as such, you need to make duplicate copies of some system files and change some of their names. (You also need to set up the SRM system, as described in a later section of this chapter.)

This type of configuration usually requires only the following type of modification to the standard configuration:

- Change file names and copy them to different volumes

You can optionally make this type of configuration change, depending on the hardware available and the desired use of it by the different systems you want to have on-line:

- Add module(s) to INITLIB

This type of change does not usually require changes to the TABLE program.

See the discussion of Copying System Files and Changing Their Names in the Modifying the Configuration section for further details.

Adding Interfaces and Peripherals

Here is a brief summary of how to add several interfaces and peripheral devices to your system.

Hardware Configuration

You should configure each interface according to the instructions given in its installation manual. Most switches can be set to their factory defaults; however, the Pascal documentation will tell you when you will need to change the switch settings from the defaults.

Software Configuration

Using a peripheral device (and the corresponding interface) for File System operations may require this type of change to the standard configuration:

- Add module(s) to INITLIB

You may need to (or optionally want to) perform this type of configuration change:

- Modify the TABLE program

Here is a list of interfaces and peripheral devices and corresponding configuration modifications you will need to make in order to use each one. See the discussion of the type of change that you will make in the Modifying the Configuration section.

- **HP 98620 Direct Memory Access (DMA) Interface**
  The driver for this interface is module DMA, which is present in the original INITLIB. The interface is always used in conjunction with other cards.
- **HP 98622 GPIO (16-bit parallel) Interface**
  To drive this interface with the IO Library, add module GPIO. (GPIO is **not** required if you use the interface only for the HP 9885 disc; however, F9885 is required.) See the *Pascal 3.0 Procedure Library* manual for further details regarding I/O applications.

- **HP 98624 HP-IB Interface**
  Using this HP-IB interface requires module HPIB, which is already present in supplied INITLIB. See the *Pascal 3.0 Procedure Library* manual for further details regarding I/O applications.
- **HP 98625 High-speed Disc Interface**
  This is a form of HP-IB interface, but it is only for use with discs and, oddly enough, printers. Modules DMA (already present in the standard INITLIB file) and DISC_INTF (not in the standard INITLIB) are required to use this interface.
- **HP 98626 Serial RS-232 Interface**
  To drive this interface, install module RS232. See the *Pascal 3.0 Procedure Library* manual for further details regarding I/O applications.
- **HP 98627 Color Output Interface**
  Using this interface is described in the *Pascal 3.0 Graphics Techniques* manual for further details regarding I/O applications.
- **HP 98628 Data Communication Interface**
  To drive this interface, install module DATA_COMM. See the *Pascal 3.0 Procedure Library* manual for further details regarding I/O applications.
- **HP 98629 SRM Interface**
  Using this interface requires that you set-up an SRM system. See the Setting Up the SRM System and the Example SRM Configuration sections of this chapter for further details.
- **HP 98630 Breadboard Card**
  This interface is intended for use only by system designers.
- **HP 98635 Floating-Point Math Card**
  No additional modules are required to use this card. However, you will need to use one of the FLOAT_HDW Compiler options; see the Compiler chapter for further details. The FLTLIB:F-GRAPHICS module is optimized for use with this card.
- **HP 98644 Serial RS-232 Interface**
  To drive this interface, install module RS232. See the *Pascal 3.0 Procedure Library* manual for further details regarding I/O applications.
- **Printers**
  Module PRINTER (already present in standard INITLIB) is required to drive all printers ("local" printers, not those on SRM), regardless of the type of interface being used. Additionally, module HPIB (already present in INITLIB) is required for HP-IB printers. Printers with RS-232C interfaces can be used with the HP 98626 RS-232C Serial interfaces if module RS232 is added to INITLIB. To drive a printer with an HP 98628 Datacomm interface, you will need to add module DATA_COMM.

  If a printer with an RS232C interface is to be recognized by the File System (for example, volume PRINTER:), then you will need to modify the TABLE program. See the Changing the System Printer section for further details.

- **Graphics Output and Input Devices**
  To talk to "local" (i.e., non-SRM) HP plotters via HP-IB requires module HPIB (already present in the supplied INITLIB). Modules DATA_COMM and SRM are required if you are using the plotter spoolers on an SRM system. In addition, you will need to use modules in the GRAPHICS library. Normally, you will not access any local plotter through the File System (i.e., you will not access it through a logical unit number); thus, you will not need to add modules to INITLIB or modify the TABLE program. See the *Pascal 3.0 Graphics Techniques* manual for further details.
- **Mass Storage Devices**
  You will almost always access mass storage devices (such as disc and tape drives, EPROM, and Magnetic Bubble memory) from the File System. The TABLE auto-configuration program finds most "common" disc peripheral devices; however, to use "non-standard" devices like EPROM and Bubble cards, you will need to modify the program. See the corresponding sections of this chapter for further details.

HP-IB Disc Performance Considerations

Disc performance is primarily determined by the device itself, but it may also be affected by the hardware used to interface the disc to the computer. Three common interface usages and their relative performance is given in the table below.

Lowest	Internal HP-IB or HP 98624 HP-IB interface (without a DMA card)
Higher	Internal HP-IB or HP 98624 HP-IB interface (with a DMA card)
Highest	HP 98625 High-Speed Disc interface and an HP 98620 DMA card (the 98625 card *requires* a DMA card)

The 913xA Hard Discs Drives (excluding the V, B, and XV suffix drives) show an increase in performance when a DMA card is used.

Although it is not required, you should use an HP 98625 High-Speed Disc interface with CS80 discs for optimal performance.

While the HP 9121, 9895, 9133 and 9134 discs can be used with the HP 98625 High-Speed Disc interface, they do not realize any increase in performance.

Note

Never use the HP 98625 High-Speed Disc Interface with an HP 82901, 82902, or 9135 disc drive.

Setting Up an SRM System

The Shared Resource Management (SRM) System is a "file server" system that allows several workstation computers to share file-oriented devices like disc drives, printer spoolers, and plotter spoolers. Also, the SRM may be the only mass storage device for a machine with no local disc drives.

This section only briefly explains what is required to configure Pascal workstations in order to access an SRM system. Here are the main steps:

1. Add modules DATA_COMM and SRM to INITLIB and re-boot, or execute them and re-execute TABLE; this step provides minimal access to the SRM through unit #5:
2. Copy files to certain SRM directories, and optionally re-name files; this step allows you to use unit #45: as the system volume and to boot from an SRM (if your computer is equipped with Boot ROM 3.0 or later)
3. Modify the TABLE program, and re-execute it; this step allows you to assign additional unit numbers to the SRM system

Because configuring a Pascal workstation to access an SRM system is not a trivial task, it is used as the example special configuration. See the subsequent section called An Example SRM Configuration (near the end of the chapter) for further details.

Changing the System Printer

Normally, the TABLE program assumes that the "system printer" (the PRINTER: volume, unit #6) is an HP-IB device at select code 7 with primary address 01. This section tells what is required to override this assumption. Here are the general changes you will need to make:

- If the printer is not an HP-IB device, you may need to add the corresponding driver module(s) to INITLIB. See the Adding Modules to INITLIB discussion in the subsequent Modifying the Configuration section.
- Modify the TABLE program so that it sets up the printer as the system printer (volume PRINTER:). See the Local Printer Type Option discussion in Modifying the TABLE Program.

Setting Up Printers with RS-232C Interfaces

The TABLE source program provides a very clean way to set up an RS-232C printer as the PRINTER: volume. Here are the conditions required:

- The RS-232C interface can be an HP 98626 or 98644 RS-232C Serial, a built-in RS-232C Serial, or an HP 98628 Datacomm interface. The default select code is 9, but you can change the `dav` variable (device address vector) to use another select code.
- In order to use the 98626, 98644, or built-in serial interfaces, you will need to add module RS232 to INITLIB
- In order to use the 98628 interface, you will need to add module DATA_COMM to INITLIB.

- The factory default switch settings[1] for these interfaces are as follows:

  Interrupt level set for level 3
  Baud rate set for 2400 baud
  Stop bits switch set for 1 stop bit
  Bits/char. switch set for 8 bits
  Protocol set for XON/XOFF
  Parity set to off

If your printer uses other parameter(s), then set the interface card[2] to match your printer. See the interface's installation manual for switch locations and settings.

The select code of the interface is assumed to be 9; either set the interface to this select code or modify the `sc` parameter in CTABLE to match the select code of your interface.

You may also need to change the `local_printer_timeout` variable to match your printer's characteristics. See the Local Printer Options section in the discussion of the CTABLE program.

Using Bubbles and EPROM

Magnetic bubble memory and EPROM (erasable programmable read-only memory) are both types of non-volatile memory. The Pascal 3.0 Workstation system allows you to use HP 98259 Magnetic Bubble Memory and HP 98255 EPROM cards as mass storage devices. This section briefly outlines what is required to configure your system to use these Series 200 cards. The Non-Disc Mass Storage chapter gives further instructions.

You will normally be accessing these cards as mass storage devices. Here are the general steps required to make these devices accessible to the File System:

- Add the appropriate driver module(s) to INITLIB:

  To use a Bubble card, add the BUBBLE module to INITLIB. This module adds both read and write capabilities for Bubble cards to the system.

  To read EPROM cards (which have already been written), add the EPROMS module to INITLIB. (To program EPROMs requires an extra step, described below.)

- Modify the TABLE program so that it assigns a logical unit number to the device(s). See the discussion of Table Entry Assignment Templates in the Modifying the TABLE Program section.

See the the Non-Disc Mass Storage chapter for the complete description of using these cards.

[1] The 98644 card has no baud rate or line control switches, so the Pascal system sets these default paramters.

[2] With the 98644 card, you may need to write a short application program to set the parameters. See the *Pascal Procedure Library* manual for details.

Using Alternate DAMs

The files on a disc are found and accessed by means of a directory which describes where the files are located, how big they are, what types of data they contain, etc. The directory is stored on the disc itself. There are many reasonable ways to organize discs, depending on one's purposes. The methods of accessing these alternative organizations are called "Directory Access Methods", or DAMs. A mass storage volume can be read or written by the File System only if the correct DAM is used.

Pascal 2.0 and later versions support three mass storage directory organizations: the Workstation Pascal 1.0 format (WS1.0, similar to UCSD format); HP's Logical Interchange Format (LIF); and the Shared Resource Manager's (SRM) hierarchical, or "structured," directory format (SDF).

In the case of Shared Resource Management discs, the DAM is supported in the SRM itself; what Pascal supports is the communication of DAM requests to the SRM. The SRM method can only be used with remote mass storage over an SRM hookup. The other two methods can be used with any local mass storage device.

The DAM used for each logical unit is selected by the TABLE configuration program. The standard TABLE selects LIF as the "primary" DAM, and UCSD (Workstation 1.0 compatible) as "secondary". (Sometimes the word "alternate" is used instead of "secondary".) The primary DAM is the one used for *blocked* units in the range of #1 through #40 (except #5, auto-configured as the SRM unit). The secondary DAM is used by blocked units in the range of #43 through #50 (with these exceptions: #45 is auto-configured as the SRM system unit, if appropriate; memory volumes always use the primary DAM).

This secondary DAM is available to allow discs with the secondary directory format to be used by Pascal programs and the Filer utility. The secondary DAM is in no way restricted from normal use by the File System; discs in the secondary DAM units can be read and written directly by Pascal programs.

If Pascal 3.0 is booted up just as shipped, the primary DAM will be LIF. In this case, Pascal 1.0 discs must be accessed through the alternate DAM units. To make the Pascal 1.0 format the primary DAM, you will need to change the TABLE program. See the section called Modifying the Configuration Table later in this chapter for further details.

CAUTION

DO NOT MAINTAIN BOTH LIF AND UCSD DIRECTORIES ON ONE LOGICAL VOLUME. THE DIRECTORIES HAVE NO KNOWLEDGE ABOUT THE OTHER'S EXISTENCE, SO EACH CAN READILY DESTROY DATA IN THE OTHER DIRECTORY.

Comparison of LIF and WS1.0 DAMs.

In both DAMs, all the space allocated to a single file is contiguous. Consequently, if the free disc space is fragmented, either DAM may be unable to create a new file of a specified size even though there is enough total free space on the disc.

In either DAM, it may not be possible to extend (append to) an existing file even if the disc volume has some free space. A file in either DAM can only be extended if there happens to be free space immediately following the file. Appending to files was not allowed in Pascal 1.0.

Letter case is significant in LIF file identifiers; for instance, the file called "Charlie" is not the same as "charlie". Letter case is not significant under the Workstation DAM (more precisely, file names are automatically converted to upper case in Workstation disc directories). The same comments apply to volume names in the two DAMs.

Workstation DAM file names may be up to 15 characters long. LIF names are restricted to 10 characters. In many cases this difference need not be a problem. Most file names used by the Pascal system end in a five-character suffix such as ".TEXT" and ".CODE"; hence the useful part of such names is 10 or fewer characters. The LIF DAM implementation encodes recognized standard suffixes into the suffix of a LIF file name, so that nine characters are available for the significant part of the name. This encoding is transparent, as is the decoding back into the full suffix when necessary.

Recommendations For Selecting Primary DAM.

If you are a new user and have no existing discs in the WS1.0 format, *we recommend that you use the system as supplied, with LIF as the primary DAM.* LIF is an HP standard for information interchange among computer systems. For instance, the BASIC and HPL systems that run on your Series 200 computer use LIF directories. The boot device must have a LIF directory unless you are booting from SRM.

If Pascal 2.0 or later system version is booted as shipped, the primary DAM will be LIF. If you have discs generated by Pascal 1.0 you have three choices.

- Change the primary DAM to the Pascal 1.0 directory.
- Adopt LIF for new volumes but access your Pascal 1.0 directories through the limited number of alternate DAM units.
- Transfer your old files on Pascal 1.0 volumes to new LIF volumes.

The choice is primarily one of convenience, although in the long run there may be some advantages to LIF. Since Pascal programs which ran under the 1.0 release must be recompiled to run under Pascal 3.0, you may choose to convert your discs as well.

Moving Files Between WS1.0 and LIF Volumes.

The following steps outline the method of moving files from one directory type to another.

1. Put the ACCESS: disc in a disc drive and press F to run the Filer.
2. Put the source disc in a drive configured for its type of DAM, and the destination disc in a drive configured for its DAM. (See below)
3. Use the Filer's Filecopy command to move files from one disc to the other. The Filer commands will work with either DAM.

Note that the alternate DAM units allow either DAM to be used in the same drive. For instance, you can copy a file from #43 to #3, both of which are assigned to the right-hand flexible disc drive in a Model 236 or 226 computer. For example:

Press F (for Filecopy), then enter:

```
#43:CHARLIE.TEXT,#3:$
```

The Filer will tell you to when to swap discs.

Remember that the name of a file in a WS1.0 directory may be too long for a LIF directory. You may have to invent a shorter name.

By the way, it's a good idea to develop the habit of using uppercase letters in the names of LIF files, because some other systems will not allow or recognize file names with lowercase letters.

Note that directories created by Pascal 1.0 have either 77 or 233 entries, whereas WS1.0 directories created by Pascal 2.0 and 3.0 have a variable number of entries specified by the user. Thus you can use Pascal 2.0 and 3.0 to create WS1.0 format directories which aren't readable by Pascal 1.0; whereas all Pascal 1.0 directories are readable by Pascal 2.0 and 3.0.

Modifying the Configuration

This section describes the mechanics of modifying the "standard" configuration methods of the system as it was shipped to you. Here are the general methods of modifying the standard configuration:

- Coalescing adjacent hard disc volumes
- Copying system files and changing their names
- Using AUTOSTART and AUTOKEYS Stream files
- Adding driver modules to INITLIB
- Modifying the standard TABLE program

Coalescing Hard Disc Volumes

As discussed previously, you can manually coalesce adjacent logical volumes on hard discs. For instance, suppose that you have an HP 9133V hard disc drive which is partitioned into the standard 4 logical volumes; the standard volume size is approximately 1 Megabyte. However, you want to increase the size of the first logical volume. You can easily coalesce the second volume with the first to double the size of the first. (This type of change is *only* possible with Option 10 machines; machines without this option cannot be logically partitioned.)

Overview of the Example Procedure

To coalesce the two logical volumes in this example, here are the steps you will take.

1. If the disc has not been initialized, then you will need to do so before continuing with this procedure.
2. Invalidate the directory of the second volume by overwriting it with the data in a file. (Since the purpose of this step is to invalidate the directory, this file must *not* resemble a directory.)
3. Change the Unit Table by running the standard TABLE program. TABLE will find the invalid second directory, invalidate the corresponding Unit Table entry, and enlarge the volume size parameter of the preceding Unit Table entry (the first volume's Unit Table entry). This step sets up the Unit Table in preparation for coalescing the two volumes. Note that the first volume's directory *on the disc* has *not* been changed at this point; it is still the original size.
4. Create a new directory on the disc for the first volume; this directory will reflect its new size. To do this, you will need to destroy the first directory on the disc and then use the Filer's Zero command to zero it. The Zero command will read the size for the first volume *from the Unit Table*, since it will not have found a valid directory on the disc. The two volumes will then be "coalesced" *on the disc* when the first directory is enlarged as it is zeroed.

Prerequisites

You should perform this operation *before* placing any valuable data in the volumes to be coalesced; however, if you have already used the volume, then you can back-up these files on another volume (such as a floppy disc or another hard disc drive). Once a volume has been coalesced with another, any data in it *cannot* be accessed.

The Example

The following procedure is an example of coalescing the second volume of an HP 9133V hard disc with the first volume, which results in approximately a 2-Megabyte first volume; the original third and fourth volumes will be left at the default size of approximately 1 Megabyte.

1. If the disc to be partitioned (here, the 9133V) has not already been installed with switches set properly, do so now. Set the drive's HP-IB address to a value that will ensure that it has a high enough priority to be assigned unit numbers. (Device Priority is fully discussed in The Booting Process at the beginning of this chapter.) For this example, we will assume that it will be assigned unit numbers 11 through 14.
2. If the disc has not already been assigned unit numbers (such as during a previous boot sequence), then use the eXecute command to run the standard TABLE program. If this program is not currently on an on-line volume or P-loaded into memory, then you will need to insert the BOOT: disc into a drive. Press (X) at the Main Command Level. The system prompts with this question:

   ```
   Execute what file?
   ```

   Enter the file specification of TABLE; the following specification indicates that it is on the BOOT: volume:

   ```
   BOOT:TABLE.
   ```

   The trailing period is required to suppress the otherwise automatic ".CODE" suffix, since this file's name on the disc has no suffix.

   When TABLE has finished, the disc should have been found and assigned unit numbers (we will assume 11 through 14). However, if it has not been previously initialized and directories zeroed, then unit numbers assigned to it will not show up in a Filer's Volumes command (they are invalid because the corresponding directories were found to be invalid).
3. At this point there are two main situations possible: the disc either has or has not been initialized.

   a. If it has *not* been initialized, do so now; proceed with step 4.

   b. It it has already been initialized, you have two more alternatives.

   If volumes have been coalesced and you want to split them (or if it is a disc initialized as one large volume), then you will need to first partition it into smaller volumes. Proceed with step 5.

   If it is has already been initialized but is still partitioned into the default number of volumes (with default sizes), or if it has volumes which have been coalesced and you don't need to split them, then you can proceed to step 6.

4. If the disc has *not* yet been initialized, then you will need to initialize it now.

   Since the disc has not been initialized, the standard TABLE program will not have found *any* valid directories at expected locations on the disc, and therefore will assume that it is to be partitioned into the default number of volumes (shown in the discussion of partitioning given in The Booting Process early in this chapter). It will build the Unit Table accordingly.

   a. Put the ACCESS:MEDIAINIT.CODE program on-line, and press X to execute it. The system responds:

      ```
      Execute what file?
      ```

      If the program is on the ACCESS: disc, enter:

      ```
      ACCESS:MEDIAINIT
      ```

      The ".CODE" suffix will automatically be appended to the file name.

      The program prompts for a unit number:

      ```
      Volume ID?
      ```

      Enter the unit number of the first volume on the disc that is to be initialized. In this case, enter:

      ```
      #11:
      ```

      The program asks for verification:

      ```
      Device: 913xA series hard disc, 707, 0
      Logical unit #11 - < no directory >

      WARNING: the initialization will also destroy:
        #12: < no dir >
        #13: < no dir >
        #14: < no dir >

      Are you SURE you want to proceed? (Y/N)
      ```

      The `913xA` corresponds to the 913x "V" suffix drives. The select code and HP-IB address (`707`), and drive number (`0`) should also correspond to the disc to be initialized. If not, then answer N and correct the problem. If this is the disc you want to initialize, then answer affirmatively by pressing Y. The program then displays:

      ```
      Medium initialization in progress
      ```

      The program takes about 15 minutes to initialize this type of disc.

      After the program has finished, it displays this message:

      ```
      Medium initialization completed
      ```

      Each volume's directory is then "zeroed" (cleared, named, and validated):

      ```
      Volume zeroing in progress
      ```

      Here is the message that indicates the entire initialization and zeroing is successful:

      ```
      Volume zeroing completed
      ```

b. Verify that the volumes are have been zeroed and are accessible by using the Filer's Volumes command. Press V, and the Filer shows you the volumes currently on-line:

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 * BOOT:
  6   PRINTER:
 11 # V11:
 12 # V12:
 13 # V13:
 14 # V14:
Prefix is - BOOT:
```

b. If all volumes have been initialized and directories zeroed properly, proceed to step 6.

5. If volumes on the disc have been coalesced and you want to split them (or if the disc was initialized as one large volume), then you will need to restore part or all of the default partitioning structure now.

   To do this, you will need to destroy the existing directories on the disc which are to be split. For instance, if your disc is one large logical volume, then you merely need to destroy the first directory, since it is the only directory that currently exists on the disc.

   a. To destroy a directory on the disc, you will overwrite the directory with a file (the MEDIAINIT.CODE file will work for this purpose). Make sure that the Filer is on-line and then invoke it by pressing F from the Main Level. Then use the Filecopy command to overwrite the directory with the file; press F and the following prompt is displayed:

   ```
   Filecopy what file?
   ```

   Enter:

   ```
   ACCESS:MEDIAINIT.CODE
   ```

   It then asks:

   ```
   Filecopy to what?
   ```

   You will answer:

   ```
   #11:
   ```

   The Filer verifies with this prompt:

   ```
   Destroy directory V11 ?
   ```

   Affirm that you want to destroy the directory by pressing Y.

   The Filer then shows that it has made the requested copy:

   ```
   ACCESS:MEDIAINIT.CODE        ==> #17:
   ```

b. If other existing directories on the disc are to be split, then destroy each by repeating this process.

c. After destroying all directories to be split, run TABLE again to restore the Unit Table to set up the default partitioning. This step does **not** partition the disc; it "partitions" the Unit Table in anticipation of the subsequent disc partitioning.

d. Now partition the disc by zeroing volumes #11, #13, and #14. If you are not still in the Filer, put it on-line and press (F). Use the Zero command to zero the first volume on the disc. Press (Z). The Filer responds with this prompt:

```
Zero directory (NOT valid for SRM type units)
Zero what volume?
```

Enter the unit number of the first volume:

```
#11:
```

The Filer then responds:

```
Destroy V11 ? (Y/N)
```

Press (Y) to confirm the command. The Filer then prompts for the number of file entries to be contained in the directory:

```
Number of directory entries?
```

If you want a number other than the default (80), then enter it now; otherwise, press (Return) or (ENTER) to accept the default.

The Filer next prompts for the volume size. The number shown in parentheses is the default:

```
Number of bytes (1206272) ?
```

Accept the default by pressing (Return) or (ENTER).

Finally, you will be prompted for the new volume name:

```
New directory name ?
```

Enter any valid volume name of up to 6 characters. For this example, enter:

```
V11:
```

The Filer verifies that it has the name you requested:

```
V11: correct ? (Y/N)
```

Press (Y) to confirm the name, if it is correct. If is not, then answer (N); you will need to start the Zero command again.

If you confirmed the name, the Filer shows that the directory was created:

```
Volume V11 zeroed
```

e. Repeat this process for each unit number to Zero all directories which you want to *remain* on the disc; you need not Zero those that will be coalesced later in this procedure.

6. If the second directory exists (unit #12), then destroy it. If it does not exist, then proceed to step 7.

   To destroy a directory, use the Filer's Filecopy command to copy a file into the directory (the MEDIAINIT.CODE file will work just fine for this purpose). Make sure the Filer is on-line, and press F to enter the Filer. Invoke the Filecopy command by pressing F. It prompts:

   ```
   Filecopy what file?
   ```

   Enter the specification of the MEDIAINIT file:

   ```
   ACCESS:MEDIAINIT.CODE
   ```

   The system promts:

   ```
   Filecopy to what?
   ```

   Enter the specification of the second directory:

   ```
   #12:
   ```

   Since a directory already exists on this volume, the Filer prompts to see if you really want to proceed and destroy this directory:

   ```
   Destroy directory V12 ? (Y/N)
   ```

   Type Y to enter an affirmative response. The Filer then shows that it completed the operation by displaying this message:

   ```
   ACCESS:MEDIAINIT.CODE           ==> 12:
   ```

7. Now you should execute TABLE again. This execution of the program will find no second directory, and consequently will make the Unit Table entry for the first directory reflect the size of both first and second volumes (about 2 Megabytes). No changes to the disc will be made by this step, however.

8. Now destroy the first directory and then Zero the volume. Destroying this directory is necessary in order for the Zero command to read the size of the volume *from the Unit Table.* If it is *not* destroyed, then the volume size will be read from the disc and the volumes will *not* be coalesced; the first directory will retain its original size.

   a. Use the Filecopy command to overwrite the first directory. While in the Filer, press F. The Filer prompts:

      ```
      Filecopy what file?
      ```

      Answer:

      ```
      ACCESS:MEDIAINIT.CODE , #11:
      ```

      The Filer then asks:

      ```
      Destroy directory V11 ? (Y/N)
      ```

      Answer Y to affirm that you do want to destroy it.

b. Finally, Zero unit #11's directory. Press Z, and the Filer prompts:

```
Zero directory (NOT valid for SRM type units)
Zero what volume?
```

Answer:

```
#11:
```

The Filer then responds:

```
Destroy V11 ? (Y/N)
```

Press Y to confirm the command. The Filer then prompts for the number of file entries to be contained in the directory:

```
Number of directory entries?
```

If you want a number other than the default (80), then enter it now; otherwise, press Return or ENTER to accept the default.

The Filer next prompts for the volume size. The number shown in parentheses is the default (note that it is now twice its former size):

```
Number of bytes (2412544) ?
```

Accept the default by pressing Return or ENTER.

Finally, you will be prompted for the new volume name:

```
New directory name ?
```

Enter any valid volume name of up to 6 characters. For this example, enter:

```
V11:
```

The Filer verifies that it has the name you requested:

```
V11: correct ? (Y/N)
```

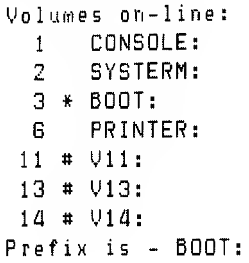
Press Y to confirm the name, if it is correct. If is not, then answer N; you will need to start the Zero command sequence again.

If you confirmed the name, the Filer shows that the directory was zeroed:

```
Volume V11 zeroed
```

9. After zeroing has completed, verify that the disc is as partitioned as desired.

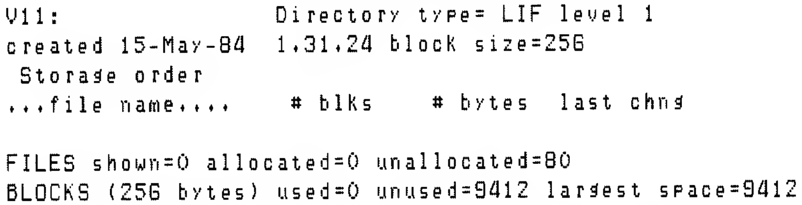
   a. Use the Filer's Volumes command to verify that there are only volumes #11, #13:, and #14: on the disc.



```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 * BOOT:
  6   PRINTER:
 11 # V11:
 13 # V13:
 14 # V14:
Prefix is - BOOT:
```

   If you don't have the partitioning scheme that you want, you may have a mistake during the procedure. You may need to repeat the appropriate part(s) of the procedure.

   b. Use the Filer's List_directory command to verify that volume #11: is now larger. Look for the number of available sectors (it should be approximately as shown below).



```
V11:                Directory type= LIF level 1
created 15-May-84   1.31.24 block size=256
 Storage order
...file name....     # blks     # bytes   last chng

FILES shown=0 allocated=0 unallocated=80
BLOCKS (256 bytes) used=0 unused=9412 largest space=9412
```

   Note that the number of entries you specified for the directory will affect the number of sectors usable for files. This example shows the number of sectors left after allocating space for 80 directory entries (files). You will have fewer sectors usable for files if you specified a greater number of entries.

   If the size is not what you expected, then you may have made a mistake during the procedure. If so, you will need to repeat the appropriate part(s) of the procedure.

Copying System Files and Changing Their Names

One of the easiest ways to change the configuration of your system is to copy files from the flexible discs on which it is shipped to discs with better performance (such as local hard discs or SRM discs). This section describes several things to consider while making this type of modification to your system.

Copying Files to the System Volume

If you have a system with one hard disc (such as a CS80 or 913x hard disc), an SS80 flexible disc (such as the 9122), or an eight-inch flexible disc (such as the 9895), then that device may be selected as the system volume during the boot process. (The system volume and its uses are described in the *Pascal 3.0 User's Guide* and in the File System and Filer chapters of this manual.) It is often useful to copy the most-used of the following "system files" to this system volume to increase performance.

```
EDITOR
FILER
COMPILER
LIBRARY
LIBRARIAN
ASSEMBLER
```

If you P-load these files, then you may not want them to take up room on your system volume; you may want to put them on another less-used volume.

As mentioned at the beginning of this chapter, the following files are used during the boot process. They are on the standard BOOT: disc shipped with your system.

```
SYSTEM_P
INITLIB
TABLE
```

If you have Boot ROM 3.0 or later versions (and not 3.0L), these BOOT: files may be placed together on any volume you choose, provided it has a LIF or Shared Resource Management (SRM) directory. They can also be renamed; the purposes of and conventions for renaming these BOOT: files will be described later in this section. If you do not have Boot ROM 3.0 or later version (which is only possible with earlier 9826 and 9836 computers), these files must all be on the right-hand internal disc drive. (The method of determining which Boot ROM you have is described in the *Pascal 3.0 User's Guide.*)

The STARTUP file is also used during the boot process, but it can either be on the boot volume or on the system volume. You might leave it on the boot volume if there isn't room on the system volume. However, if possible, put it on the system volume so that it loads faster.

If you have an HP 9885 8-inch disc drive as the system device, not all the system files will fit on it. Using the above procedure, copy onto it those system files which are used most frequently (such as EDITOR, FILER, and COMPILER).

Default BOOT: File Names

On the BOOT: discs shipped from the factory, the files used during the boot process are named as follows:

```
SYSTEM_P
TABLE
INITLIB
STARTUP
```

If these files are copied to another local (non-SRM) mass storage volume and the names are retained, they will boot normally.

Re-Naming the BOOT: Files

If you change the name of SYSTEM_P (the system Boot file), then you must also change the names of some other files on the BOOT: disc. The advantage is that the same Boot file (with different names) can load specialized BOOT: files for unique hardware configurations.

Note

The term "BOOT: file" is used to identify a file used during the boot process; these files are on the standard BOOT: disc shipped with your system.

The term "system Boot file" is used to identify a file that is found and loaded by the Boot ROM, such as SYSTEM_P; this file then loads the corresponding operating system.

The Boot ROM 3.0 and later versions also recognize system Boot files if the file name begins with "SYS". If you rename the Pascal system Boot file (SYSTEM_P), there are file naming rules you must follow so the system Boot file can identify the other BOOT: files. The rules for BOOT: file names are as follows:

- If the complete string "SYSTEM_" is used in the system Boot file name, up to the next three letters of the file name are added to the base of the other BOOT: file names (INIT, TABLE, START).
- If only "SYS" is used in the system Boot file name, up to the next seven letters of the file name are added to the base of the other boot file names (possible only with Boot ROM 3.0 and later versions).

For example, if you change the system Boot file's name from SYSTEM_P to SYSTEM_P3 (for Pascal 3.0), then the Boot file will look for the following files:

```
INITP3
TABLEP3
STARTP3
```

Keep in mind that file names on a LIF directory must be 10 characters or less.

If you change the name to Boot file's name SYS_SRM_P3, it will look for the following files:

```
INIT_SRM_P3
TABLE_SRM_P3
START_SRM_P3
```

File names on SRM directories can be up to 16 characters.

In general, SYSTEM_P could be the Pascal Boot file that loads the standard Pascal BOOT: files. SYSTEM_P3 is the same Boot file, but INITP3 and TABLEP3 could support the hard discs. SYS_SRM_P3 is the same Boot file, but INIT_SRM_P3 and TABLE_SRM_P3 could support SRM.

Normally, a special TABLE is not required for hard discs or SRM systems, although you may wish to create one for a special application.

SRM users needing a special TABLE should use the "private" system volume based on the node address mechanism to keep their own custom TABLE and INITLIB (i.e., the default system volume when booting a workstation from SRM is the /WORKSTATIONS/SYSTEM*nn* directory, in which *nn* is the node address of the workstation). See the subsequent Example SRM Configuration section for further details.

AUTOSTART and AUTOKEYS Stream Files

As discussed in a previous section, Stream files allow execution of commands just as if you had entered them from the keyboard. When you put a Stream file named AUTOSTART on your system volume, the keyboard commands the file contains are automatically executed during the booting process; if the volume is a read-only device, such as EPROM, then you should call the stream file AUTOKEYS. (The Stream command is fully described in the Main Command Level Reference section of the Overview chapter.)

You can use autostart files to perform such functions as the following: load drivers; use the What command to change the system files, system library, default or system volume; re-execute the TABLE program (or execute another like it). Be aware that there are also other ways to perform this type of configuration; however, this method can be used to quickly or temporarily change the configuration by creating different stream files, renaming the one you want to be used as the autostart file to `AUTOSTART` (or `AUTOKEYS`), and then re-booting.

In order to configure your system to access an HP 98626 RS-232C Serial interface, for instance, you can install the driver module with an autostart file. Here is an example stream file that performs this installation (this example assumes that the file named RS232 is on the volume that is chosen as the system volume at power-up):

```
<blank line>
<blank line>
x*RS232.
v
```

The two blank lines which occur first are two carriage returns in the file which are "null" responses to the Time and Date prompts at power-up. These null responses get you to the Main Command Level.

The "x" in the first column of the third line is an eXecute command. The period at the end of the file name prevents the system from appending ".CODE" to the file name. The "v" at the end of the file is the Version command. It gives you another chance to type in the time and date.

After you've created your AUTOSTART file, be sure that you store it on the system volume. This is done by Quitting the Editor, selecting the Write option, and entering this file specification:

```
*AUTOSTART.
```

The period at the end of the file name prevents ".TEXT" from being added to the file name. If you were a Pascal 1.0 user, the file was called AUTOSTART.TEXT. With Pascal 2.0 and later versions, it is called AUTOSTART. Notice that uppercase characters must be used.

Adding Modules to INITLIB

As mentioned previously, the INITLIB supplied on your original BOOT: disc contains a reasonably complete set of peripheral driver software. You may wish to install other drivers, which are supplied on the CONFIG: disc; or to conserve memory you may wish to remove items you don't need.

Unlike the System Library, modules in INITLIB are order sensitive. Certain modules, if present, must precede others in INITLIB. The list which follows shows the recommended order of all the "driver" modules supplied with Pascal 3.0. If you add or delete INITLIB modules, all the modules which are present in the resulting INITLIB should appear in the order listed.

Required Order of Modules in INITLIB

The table lists the importance of each module. Items marked "Required" are essentially required in INITLIB. Items marked "Almost" are almost always required. These modules should not be removed unless you have determined for sure they aren't needed, because they are part of the normal functioning of the system. Items marked "Development" are usually needed in a software development environment. Items marked "Optional" are optional unless required by a particular system configuration.

Required Order of Modules

Module	Where found	Importance
KERNEL	BOOT:INITLIB	Required
SYSDEVS	BOOT:INITLIB	Required
CRT	BOOT:INITLIB	Required
CRTB	BOOT:INITLIB	Required
A804XDVR	BOOT:INITLIB	Required
KEYS	BOOT:INITLIB	Required
NONUSKBD1	BOOT:INITLIB	Required
NONUSKBD2	BOOT:INITLIB	Required
BAT	BOOT:INITLIB	Required
CLOCK	BOOT:INITLIB	Required
PRINTER	BOOT:INITLIB	Development
DISCHPIB	BOOT:INITLIB	Development
AMIGO	BOOT:INITLIB	Optional
CS80	BOOT:INITLIB	Optional
IODECLARATIONS	BOOT:INITLIB	Required
HPIB	BOOT:INITLIB	Almost
DMA	BOOT:INITLIB	Development
REALS	BOOT:INITLIB	Required
ASC_AM	BOOT:INITLIB	Development
WS1.0_DAM	BOOT:INITLIB	Development
TEXT_AM	BOOT:INITLIB	Almost
CONVERT_TEXT	BOOT:INITLIB	Almost
LIF_DAM	BOOT:INITLIB	Almost
CHOOK	BOOT:INITLIB	Optional
DEBUGGER	ASM:DEBUGGER	Development
DISC_INTF	CONFIG:DISC_INTF	Optional
DATA_COMM	CONFIG:DATA_COMM	Optional
GPIO	CONFIG:GPIO	Optional
RS232	CONFIG:RS232	Optional
SRM	CONFIG:SRM	Optional
F9885	CONFIG:F9885	Optional
BUBBLE	CONFIG:BUBBLE	Optional
EPROMS	CONFIG:EPROMS	Optional
INTERFACE	CONFIG:INTERFACE	Optional
EDRIVER	CONFIG:EDRIVER	Optional
SEGMENTER	CONFIG:SEGMENTER	Optional
HPHIL	CONFIG:HPHIL	Optional
MOUSE	CONFIG:MOUSE	Optional
LAST	BOOT:LAST	Required

Individual Module Descriptions

Here are brief descriptions of each of the above modules.

- KERNEL is the "core" of the system, containing the Library facility for the Linking Loader and basic File System support. It is always required.
- SYSDEVS, CRT, CRTB, A804XDVR, KEYS, NONUSKBD1, NONUSKBD2, BAT, and CLOCK are responsible for the CRT display, keyboard, foreign character set, battery backup, and clock. They are broken out into several small modules so they may be replaced individually if desired.

  Module CRT is only needed if your computer has an alphanumeric display that is separate from the graphics display (this is the case for most Series 200 Computers). Module CRTB is only required in computers with bit-mapped graphics displays (such as the HP 9837).

  Modules NONUSKBD1 and NONUSKBD2 need only be present or replaced by code with equivalent function if foreign keyboards are used.

  Module CLOCK is only required for *programmability* of the battery back-up hardware option.

Note

You can also IMPORT the SYSDEVS module (i.e., use data objects and code declared in it), which is described in the Procedure Library manual. This is the only system module that is described fully enough in this manual set to use in this fashion.

- PRINTER is required to drive all printers, regardless of the type of interface electronics being used. It supports serial as well as HP-IB printers; however, you will have to use the RS232 driver module in order to use printers with RS-232C interfaces. You may also have to modify the variable named `local_printer_timeout` in the CTABLE program; see the discussion of modifying CTABLE later in this chapter.
- DISCHPIB, AMIGO, and CS80 modules are related. To use any external disc drive connected via HP-IB you must use the following modules:

  DISCHPIB and AMIGO for these disc models:

  9895 (8-inch flexible disc)
  9121 (single-sided 3.5-inch flexible disc)
  913x (small hard discs)
  8290x (5.25-inch flexible disc)

  DISCHPIB and CS80 for fixed discs of the Command-Set/80 (CS80) and Sub-Set/80 (SS80) disc drives, and DC600 tape drives, including these models:

  79xx (large hard discs; optional integrated DC600 tapes)
  9122 (double-sided 3.5-inch flexible disc)
  9144 (stand-alone DC600 tape drive)

- IODECLARATIONS is the lowest level of device IO support. Although it is possible to construct loadable systems without this module, only the internal disc drives on the Model 226 and Model 236 can be accessed.
- HPIB is the lowest level support for the Hewlett-Packard Interface Bus, which is HP's implementation of the IEEE-488 Standard. HP-IB interfaces include the built-in HP-IB and HP 98624 cards. Most HP peripherals have HP-IB interfaces, so you will rarely remove this module.
- DMA is the module which runs the HP 98620 Direct Memory Access interface card. DMA provides very high speed data transfers. It is also required in order to use the HP 98625 Disc Interface.
- REALS is the floating-point mathematics support package. It also supports the HP 98635A Floating-Point Math card.
- ASC_AM is the Access Method responsible for blocking and unblocking text files with the LIF-ASCII structure (.ASC files). LIF stands for Logical Interchange Format, a common file interchange structure supported by many HP computers. Since this is one of the formats used by the BASIC language system, it is a good thing to have around. It is also the format used by the SRM for spooled printer files.
- WS1.0_DAM is the Directory Access Method used by the Pascal 1.0 system, a predecessor to the one you are using. This module lets the system read and write discs in that format. Note that the WS1.0 disc organization is compatible with discs written by UCSD Pascal systems; but to read discs written by non-HP computers, a special disc driver is usually required. This DAM can be removed if you have no need to read or write discs compatible with the Pascal 1.0 system.
- TEXT_AM is the Access Method used to block and unblock text files created with the ".TEXT" suffix. These are the files normally created by the Editor (unless the user specifies otherwise). The ".TEXT" file structure is compatible with text files generated by UCSD Pascal systems.
- CONVERT_TEXT is a module used by the Compiler and other subsystems to convert among the various representations of text files. It should be present in INITLIB.
- LIF_DAM is the Directory Access Method required to read and write HP Logical Interchange Format disc directories. LIF is the primary directory organization used with Pascal 2.0 and later system versions, so this module is normally present. If you configure your system to use WS1.0 as the primary directory method (as described in the Special Configurations chapter), you may remove LIF_DAM.
- CHOOK is the driver for color monitors (such as the monitor of the HP 9836C).
- DEBUGGER is the interactive debugging tool. It is not part of INITLIB (as in pre-3.0 system versions) due to disc space and because it is a particularly dangerous thing to put in the hands of non-programmers. Module REVASM is also a handy tool to have while debugging programs; it allows you to display the contents of memory locations as Assembler language instructions (i.e., "reverse assemble" them).
- DISC_INTF and DMA modules are required in order to use the HP 98625 High-Speed Disc interface.
- DATA_COMM is the module required to drive HP 98628 Data Comm and HP 98629 SRM interfaces.
- GPIO is the module required to drive the HP 98622 GPIO (16-bit parallel) interface.

- RS232 is the module required to drive the HP 98626 and 98644 RS-232C Serial interfaces, and the built-in serial interface in the Model 216 (HP 9816) computer.
- SRM is required to drive the HP 98629 SRM interface. Module DATA_COMM is also required when using SRM.
- F9885 is required for model 9885 flexible disc drives. These discs also require the DMA module, HP 98620 DMA card, and HP 98622 GPIO interface.
- BUBBLE is the module that drives HP 98259 Magnetic Bubble Memory cards. In order to use these cards for mass storage, you will need to add this module to INITLIB and then modify the TABLE program. See the Using Bubbles and EPROMs section for further details.
- EPROMS is required to access the HP 98255 EPROM cards. In order to use these cards for mass storage, you will need to add this module to INITLIB, modify the TABLE program, and program the EPROMs using the HP 98253 EPROM Programmer card (which requires the EDRIVER module). See the Using Bubbles and EPROMs section for further details.
- INTERFACE is the interface text of all of the default modules in INITLIB. You will need to make this interface text available to the Compiler when you import any of the modules in INITLIB; you can do this by copying the module to be imported from CONFIG:INTERFACE to the System Library or using a SEARCH Compiler option that specifies the INTERFACE file. A good example is importing the SYSGLOBALS module, which requires that INTERFACE be accessed by the Compiler. See the *Pascal 3.0 Procedure Library* manual for further details.

Note

INTERFACE also contains the interface text of the SYSDEVS module. Using procedures, etc. from this module is described in the *Pascal 3.0 Procedure Library* manual. Access to most other modules in INTERFACE is not described in the current Pascal 3.0 documentation set.

- EDRIVER is the module required to program EPROMs using the HP 98253 EPROM Programmer card. In order to use EPROM cards for mass storage, you will need to add this module to INITLIB and then modify the TABLE program. See the Using Bubbles and EPROMs section for further details.
- SEGMENTER provides the ability to segment programs and run each separately. See the Segmentation Procedures chapter of the *Pascal 3.0 Procedure Library* manual for further details.
- HPHIL provides the drivers for HP Human Interface Link devices; it is an extension to the A804XDVR module. You can remove it if your computer does have not one of these devices (for example, a 46020 keyboard or a mouse input device).
- MOUSE provides a driver for the optional "mouse" input device, which can be connected to the computer through the HP Human Interface Link (HP-HIL). The driver supports using the mouse for cursor-movement input in both horizontal and vertical directions; it also defines the buttons on the mouse as (Return) or (ENTER) and (Select) ((EXECUTE)). You can access the mouse from your own applications programs; see the System Devices chapter of the *Pascal Procedure Library* manual for details.

- LAST is required in every case, and *must* be the last module in INITLIB. The purpose of this module is to actually start the system running after the contents of INITLIB have been loaded and installed in memory. LAST principally does two things: it loads and executes the configuration file called TABLE; and it loads and executes a file called STARTUP, which is usually the Command Interpreter but may be a user program.

Steps for Adding Modules to INITLIB

For this example, we will add module RS232 (on the CONFIG: disc) to the INITLIB file that you are now using to boot your system (possibly on the BOOT: disc); actually, you will create a new INITLIB that includes all existing drivers plus these additional driver modules. These are the modules required to access the HP 98626 and 98644 RS-232C Serial interface cards.

Here is an outline of the procedure that you can use for adding driver modules to the INITLIB library file. It is a straight-forward usage of the Librarian.

1. Set up mass storage. You will need enough on-line mass storage to store *two* copies of the INITLIB file: one for the source (existing) copy, and one for the destination (new) copy. This requirement is made because the new copy of the INITLIB file must not be taken off-line during the whole process.

   To satisfy this requirement, you will need one of the following configurations: one disc large enough to store both (such as a hard disc or SS80 flexible disc); two flexible disc drives; a flexible disc drive and a memory volume.
2. Copy all modules except LAST from the source INITLIB file onto the destination disc. (You may also remove modules from it as it is copied, if desired.)
3. Add the RS232 module to the INITLIB file on the destination disc.
4. Copy the module named LAST from the source INITLIB to the destination file.
5. Replace the existing INITLIB with the new file.

Mass Storage Requirements

As shipped, the INITLIB file requires about 750 sectors (190 Kbytes) of disc space. Since you will have two copies of this file on-line, and one cannot be removed during the process of adding modules to it, here are the mass storage requirements:

- If you are using small-capacity flexible disc drives (approximately 270 Kbytes per disc), then you will need either two drives or one drive and a memory volume.
- If you are using an SRM shared disc, or have a local hard disc (all have volumes with capacities of 1 Mbyte or greater) or an SS80 flexible disc (approximately 630 Kbytes per disc), then you will need only one disc.

You may also need to initialize one or two blank discs (with the MEDIAINIT.CODE utility) then label them. Disc initialization is discussed in the *Pascal 3.0 User's Guide.* Write the name on a label before applying the label to the disc; sharp instruments are likely to damage the disc. You will probably also want to make a back-up copy of the BOOT: disc on which the INITLIB file resides.

Making a Memory Volume

If you only have one small-capacity flexible disc drive, then you will need to make sure that you have enough memory to make a memory volume of sufficient size. Here is how to determine the amount of memory in your computer. If the machine is on, turn it off, along with any disc drives to which it is connected. Open the doors of any built-in flexible disc drives. If the SRM is already connected, remove the cable connected to the 98629A Resource Management interface in the back. Now turn the computer on. After going through self-test, the CRT will display the amount of available memory. If you have at least 524 000 bytes, there is "enough" memory to proceed with only one small-capacity disc drive. After turning off computer power, you can reconnect cables and then turn on any peripherals connected to your computer.

If you determined that you have "enough" memory and must use some memory for mass storage, the following steps are necessary.

1. At the Main Command Level, press M. The computer responds:

   ```
   *** CREATING A MEMORY VOLUME ***

   What unit number?
   ```

2. Enter:

   ```
   #50
   ```

   The computer then asks:

   ```
   How many 512 byte BLOCKS?
   ```

3. Enter:

   ```
   520
   ```

   The computer asks:

   ```
   How many entries in directory?
   ```

4. You answer:

   ```
   8
   ```

   The computer finishes:

   ```
   #50:  (RAM:) zeroed
   ```

5. Use the Filer's Change command to re-name the memory volume from RAM: to DEST: (which is the volume name assumed in the following procedure).

This has reserved 266 240 (520*512) bytes of memory to use as a mass storage device. It is like having a 5.25-inch disc drive with a disc named DEST: inserted in it.

Assumptions Made During this Procedure

Now you are ready to start the process of making the new INITLIB file. This procedure makes the following assumptions:

- The existing INITLIB file is on the BOOT: disc.
- The destination volume is called DEST:.
- The RS232 driver module is on the CONFIG: disc.

The Procedure

1. Make sure that the Librarian is on-line (or insert the ACCESS: disc into the drive you have been using) and press `L` to load it.
2. When you see the Librarian's prompt line at the top of the CRT, press `O` to specify the name of the (Output) file the Librarian will be creating. Enter this name as the destination (new library's) file name:

   ```
   DEST:INITNEW
   ```

   > **Note**
   >
   > If you are using a flexible drive, you must not remove the Output disc until the end of this procedure, after you have Quit the Librarian.

3. Press `I` so you can specify an Input file, then enter:

   ```
   BOOT:INITLIB.
   ```

   If the file is not on the BOOT: volume, then you will need to change the leading volume specification. Be sure to type the period after the word INITLIB in this command (to suppress the otherwise automatic .CODE suffix). The Librarian will respond by showing INITLIB as the name of the Input file.
4. Near the bottom of the screen you will see a line which says:

   ```
   M input Module: KERNEL
   ```

   Press `T` to transfer this module to the output file. After a few moments, the name of the next module will appear (probably SYSDEVS). Each time a new module name appears, press `T` to transfer it to the Output file. You should continue copying modules until the name LAST appears. **Don't copy module LAST yet.**
5. Now you must get the required RS232 drivers from the CONFIG: disc and transfer them to the Output file. If the CONFIG: disc (or a disc containing the RS232 module) is not currently on-line, then put it on-line. Press `I` for an Input file and enter this file specification:

   ```
   CONFIG:RS232.
   ```

   (If you are not copying the module from the CONFIG: disc, then use the volume specification of your source disc.) Don't forget the period after the file name.
6. When the module name RS232 shows up near the bottom of the screen, press `A` which tells the Librarian to transfer All the modules in the file. Remove the CONFIG: disc after the module has been transferred.

7. Put in BOOT: once more, press I for Input, and enter the file specification of the original INITLIB file:

   ```
   BOOT:INITLIB.
   ```

8. When module KERNEL shows up near the bottom of the screen, select module LAST instead by pressing M for module and enter:

   ```
   LAST
   ```

   Then transfer LAST to the Output file by typing T.
9. You now have all the modules in your new library. "Keep" it by typing K.
10. You may want to verify that these modules are in the new library. Press I and specify the new library as the Input file:

    ```
    DEST:INITNEW
    ```

    (The .CODE will automatically be appended to the file name.) Step through the file with the space bar. If all modules are present, then Quit the Librarian by typing Q.
11. Remove the old copy of INITLIB from the BOOT: disc with the Filer's Remove command (if the Filer is not on-line, you will need to put it on-line before trying to invoke it). Then Krunch the BOOT: disc so that you will have enough room on the disc to store the new, larger copy of the INITLIB file (INITNEW.CODE).
12. Use the Filer's Filecopy command to copy the new library file (DEST:INITNEW.CODE) onto the BOOT: disc, changing the name INITNEW.CODE to INITLIB as you copy it.
13. The next time that you boot your system with this new INITLIB, module RS232 will automatically be installed.

Modifying the TABLE Program

This section first describes the structure of the TABLE program. If you want to change something that it does, you will need to edit and re-compile the CTABLE.TEXT source program on the CONFIG: disc.

Overview of General Steps

Here are the general steps you should take to modify TABLE:

1. The Pascal source of TABLE, called CTABLE.TEXT, is provided on the CONFIG: disc distributed with every copy of the system. Read the commentary on the CTABLE source program in this section. You should follow along in the source program as you read the corresponding commentary.
2. Make your modifications to a *copy* of the program – **not the original**.
3. Compile this modified program, which yields an object code file (for instance, MYTABLE.CODE).
3. Execute the modified program to see if the results are correct. In fact, the Unit Table can be reconfigured any time by executing a version of TABLE.
4. When you are quite sure the new TABLE program is correct, use the Filer to copy the compiled code to your BOOT: disc. The name of the copy must be TABLE (not TABLE.CODE) in order to be recognized during boot-up (with Boot ROM 3.0 and later, it may begin with the letters TABLE and end with letters that match the other BOOT: file names; see the discussion of Renaming the BOOT: Files earlier in this section).

CAUTION

BE CAREFUL HERE! IF THERE IS NO BACKUP COPY OF THE BOOT: DISC CONTAINING THE ORIGINAL TABLE, AND THERE IS SOMETHING WRONG WITH THE MODIFIED TABLE, YOU MAY NOT EVEN BE ABLE TO USE YOUR SYSTEM.

5. Depending on the size of INITLIB, there may not be much room on the BOOT: disc. You may need to Krunch the disc with the Filer to make space. The modified TABLE can also be made considerably smaller by linking it to itself. This combines all the internal modules into a single module, and gets rid of module interface text and internal reference information. The procedure for linking the modules of a file is presented later in this section.

Commentary on the CTABLE Program

CTABLE is a long program; for ease of study, here is a summary of its structure. You will probably want to print out the source code and examine it in detail.

```
program ctable;

    module options;
        {Contains declarations which MAY BE EDITED to override
        many of the system defaults.}

    module ctr;          {DON'T MODIFY THIS MODULE.}
        {Exports the table entry assignment routines, which contain
        information highly specific to HP peripheral devices.}

    module BRstuff;      {DON'T MODIFY THIS MODULE.}
        {Figures out which device was the boot device.}

    module scanstuff;    {DON'T MODIFY THIS MODULE.}
        {Contains code which asks each device to identify itself.}

begin

    initialize hardware (interfaces, etc.).

    assign default 'device address vectors'.

    scan for devices on various HPIB addresses.

    scan for an internal mini-floppy drive.

    determine the nature of the boot device.

    create temporary unit table.

    make 'standard' assignments #1:-#6:
      (in temporary Unit Table).

    assign units #7:-#10: (to 2nd and 3rd priority floppies).

    assign units #11:-#40: (to local hard discs).

    assign units #41:,#42: (to tape drives).

    assign units #43:-#44: (as alternate DAMs for #3:-#4: floppies).

    assign unit #45: as additional entry for SRM
       (note the template for #46).

    assign units #47:-#50: (as alternate DAMs for #7:-#10: floppies).

    make optional templates for 'manually' overriding preceding defaults
      [hard-disc partitioning, tape drives, EPROM and Bubbles, etc.].

    copy temporary Unit Table to actual system Unit Table.
    prefix directory on SRM for #5: (default) and #45: (system).

    remove extraneous local hard disc entries if necessary.

    assign unit for system volume.

    set prefix of SRM default volume.

    re-open the 'standard' system files (#1:, #2:, and #6:).

  end.
```

Note

You may want to read through the following discussion of the standard TABLE in its entirety before editing the CTABLE.TEXT source file. On the other hand, you may want to edit it as you read the discussion.

The general recommendation is that you should edit the main program **only** if the desired result cannot be obtained by modifying the declarations in module `options`.

Modifying Module OPTIONS

This module consists only of declarations of exported types and constants. The constants are used by the main program; each section describes their effects and how to modify them.

Power-Up System Volume

The following constant selects the system volume at power-up.

```
  {power-up system unit}
    const
      specified_system_unit =
        0; { <>0 overrides auto-assignment}
```

When `specified_system_unit` is zero (the default), the program makes its own choice according to the algorithm described in the preceding discussion of The Booting Process.

If you change this constant to a non-zero value, then it indicates which of the 50 units is to be the system volume. For instance, if you change this constant to 3, then drive #3: will become the system volume. This explicit choice overrides any units specified in the subsequent `{ system unit auto-search declarations }` section of this module.

Primary Directory Access Method (DAM)

The following constant selects the primary Directory Access Method for *local* (i.e., non-SRM) mass storage devices.

```
{local mass storage directory access method}
  type
    lms_dam_type = {local mass storage dam}
      ( LIF, UCSD );
  const
    primary_lms_dam =
      LIF;
```

`LIF` selects HP's Logical Interchange Format directory; `UCSD` specifies the default format used in Pascal 1.0. The standard TABLE expects that remote mass storage devices will use the Shared Resource Manager's hierarchical (structured) directory format (SDF), so no constant declaration is needed.

Floppy Disc Unit Pairs

The standard TABLE program is set up to assign unit numbers to up to three dual floppy disc drives. They will normally occupy units in pairs (#3: and #4:, #7: and #8:, and #9: and #10:). Hard discs usually begin at unit #11: and can be assigned up to 30 unit numbers (up to unit #40:).

```
{floppy/harddisc unit number slot tradeoff's}
  const
    floppy_unit_pairs =  {[1..10]}
      3;
    harddisc_first_lun = {do not edit!}
      7+(floppy_unit_pairs-1)*2;
    harddisc_last_lun =
      40;
```

In order for the TABLE program to assign unit numbers to other than three floppy-disc pairs, you will need to change the `floppy_unit_pairs` variable accordingly. Note that changes to this constant affect the beginning unit assigned to the highest priority hard disc in the system. For instance, changing the constant to 2 causes hard discs to begin at #9, while changing it to 4 causes hard discs to begin at unit #13.

Local Printer Type Option

This constant determines the type of the local printer; it can be either HPIB or RS232.

```
{local printer type option}
  type
    local_printer_type = (HPIB, RS232);
  const
    local_printer_option = HPIB;
```

Local printers with RS-232C interfaces will **not** be recognized by the standard TABLE program. If option `RS232` is chosen, you must have an HP 98626 or 98644 RS-232C Serial interface or an HP 98628 Datacomm interface present. In order to use one of these Serial interfaces, module RS232 must be installed; using the Datacomm interface requires module DATA_COMM.

Here are the default interface switch settings:

Select code 9
Interrupt Level set to 3
Baud rate set for 2400 baud
Stop bits set for 1 stop bit
Bits/character set for 8 bits
Protocol set for XON/XOFF
Parity set to off

If your printer uses a different parameter, then you should change the interface switch setting[1] accordingly; see the interface's installation manual for switch locations and settings. If you want to use another select code, then you will need to modify the `local_RS232_printer_default_dav` parameter accordingly; see the subsequent discussion of Default Device Address Vectors.

Local Printer Timeouts

This constant determines the timeout parameter for local printers:

```
const
  local_printer_timeout =
    $IF local_printer_option=HPIB$
       12000;  {milliseconds}
    $IF local_printer_option=RS232$
       0 ;     {infinite}
```

This governs the byte-transfer timeout used by the local printer driver. The timeout, expressed in milliseconds, specifies the maximum time allowed for each byte handshake to complete. A value of zero is a special case, specifying an infinite timeout. See the commentary above this constant declaration in the CTABLE.TEXT source program for recommended values.

The policy of enforcing a timeout on each individual byte works quite well with most HP-IB printers, since they tend not to hold off bus handshakes much longer than the time it takes them to print a single character. However, with printers on other interfaces (notably serial interfaces) we have a different matter. Some serial printers will "buffer up" bytes at high speed until their internal buffer is full, but then will not allow any more tranfers until their internal buffer is almost empty. Thus, depending upon the printer's internal buffer size, the maxmimum time between two bytes being transferred may be the time it takes to print hundreds or even thousands of characters! For these printers, you might consider a timeout of several minutes, or even an infinite timeout.

In general, most HP-IB printers accept hundreds of bytes per second, so you might think that the default 12 second timeout is excessive. We were forced to use this large a number since some low-cost HP-IB printers take 8-10 seconds to execute a full-page formfeed. If you are using a faster printer, you might consider reducing the timeout to 2-3 seconds, so that a real timeout condition will be detected more quickly.

[1] With the 98644 card, you may need to write a short application program to set the parameters. See the *Pascal Procedure Library* manual for details.

Default Device Address Vectors

Here are the default "device address vectors" for devices that cannot be found by interrogation.

```
{default dav's for devices not found by scanning}
  type dav_type = {device address vector}
    packed record
      sc, ba, du, dv: -128..127;
    end;
  const
    HP9885_default_dav =
      dav_type[sc:12, ba:-1, du:0, dv:-1];
    SRM_default_dav =
      dav_type[sc: 21, ba: {node} 0,
             du: {unit} 8, dv: -1];
    BUBBLE_default_dav =
      dav_type[sc: 30, ba: 0, du:  0, dv:  0];
    local_HPIB_printer_default_dav =
      dav_type[sc:  7, ba: 1, du: -1, dv: -1];
    local_RS232_printer_default_dav =
      dav_type[sc:  9, ba: 1, du: -1, dv: -1];
```

The device address vector, or dav, is the data type which describes how a peripheral device is addressed. These constants set up the addressing which is normally used to talk to some standard peripheral devices (some of the information will be overridden if the peripheral is found at a different address).

- sc is the interface select code. Select code 7 corresponds to the built-in HP-IB port at the back of Series 200 computers. The HP 9885 disc is connected using a 16-bit parallel interface on select code 12, and a DMA card. The SRM interface is normally set to select code 21.
- ba is the HP-IB primary address of the peripheral. Usually an 8290x is addressed as device 0; so is a 9895. The 913x family of hard discs are expected (though not required) to be at primary address 3, and printers at address 1.

  For the SRM only, ba indicates the node number of the SRM interface in a cluster (as opposed to the node number of the Workstation itself).
- du selects the disc unit in a multi-drive machine. For instance, a 9121D has drives 0 and 1. With SRM systems that contain multiple disc drives, this parameter selects which disc is to be accessed.
- dv selects a particular volume in a multi-volume CS80 (7908 family) disc.

Hard Disc Partitioning

The next section that you will come to in the CTABLE.TEXT source program concerns hard disc partitioning. However, in order to be able to wisely decide whether or not you will need to modify any of these parameters (and, if so, to choose which parameter you want), you need to fully understand how partitioning works.

> **Note**
>
> If you haven't read the discussion of hard disc partitioning in The Booting Process discussion, you should do so now.

The standard TABLE program assumes that local hard discs are to be partitioned into several "logical volumes," each of which is to be assigned a unit number. The following equation conceptually describes how TABLE determines the number of volumes into which the disc is to be logically divided (the actual equation actually used is slightly more complex, because it partitions on track boundaries):

`nvols` = disc capacity DIV `mvs`

The values of `nvols` for each type of hard disc, as calculated by this equation, are shown near the end of the main part of the CTABLE program, following the comment:

```
{ templates for "manually" specifying mass storage table entry assignments }.
```

The `mvs` parameter is a constant in the `options` module.

```
{local hard disc partitioning parameters}
  type
    pp_type = {partitioning parameters}
      record
        mvs: integer;   {min vol size in bytes}
        mnv: shortint;  {max number of volumes}
      end;
```

<Comments in program about values and effects of `mnv` parameter.>

```
    const
      min_size = {in bytes [1..maxint]}
        1000000;
      max_vols = {[-30..30]; <0 means auto-coalesce}
        -30;
      HP913X_A_pp =
        pp_type[mvs: min_size, mnv: max_vols];
      HP913X_B_pp =
        pp_type[mvs: min_size, mnv: max_vols];
      HP913X_C_pp =
        pp_type[mvs: min_size, mnv: max_vols];
      CS80disc_pp =
        pp_type[mvs: min_size, mnv: max_vols];
```

The constant min_size indicates that no logical volume is to be smaller than one million bytes. The constant max_vols indicates that no device is ever to be partitioned into more than 30 logical volumes; the negative value of max_vols indicates that logical volumes that do not have valid directories are to be "coalesced" with the last preceding volume found to have a valid directory. These constants are assigned to the mvs and mnv constants for each class of device. You can change them if desired; the values and corresponding effects of mnv are described in the comments in the CTABLE program.

> **Note**
>
> The constant max_vols must **not** be greater than 30.
>
> The HP913X_A corresponds to all 5-Mbyte HP 913x Option 10 "A" drives and "V" drives with a single "disc unit" or "drive number" (as opposed to non-Option 10 "A" drives which have 4 drive numbers).
>
> The HP913X_C corresponds to all 15-Mbyte HP 913X "XV" drives.

Example of Standard Partitioning

In order to better understand partitioning, let's look at how the standard TABLE program partitions an HP 7908 hard disc. You can see most of the default parameters by looking in the templates section of the main program that begins with this comment:

```
{ current CS/80 discs "soft" partitioned by the host }.
```

These discs have a capacity of about 16 Mbytes, so nvols is 16 for this type of disc.

The size of each logical volume is given by this equation:

vol_bytes = tpm DIV nvols * bpt

in which:

vol_bytes = size of volume (in bytes)

tpm = number of tracks per disc media (device-dependent)

nvols = number of volumes expected on the disc (device-dependent)

bpt = bytes per track (device-dependent)

The last volume on the disc may contain some additional bytes according to the remainder of the above integer division:

Last volume = vol_bytes + (tpm MOD nvols) * bpt

Here are the values of the preceding parameters for an HP 7908 disc drive (they are contained in the main body of the program, near the end):

tpm = 5 * 370 { 5 surfaces with 370 tracks/surface }

nvols = 16

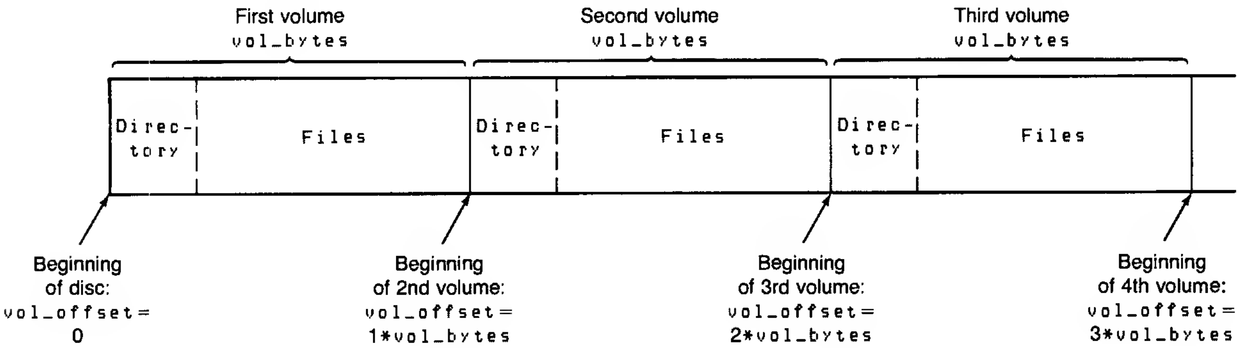
bpt = 35 * 256 { 35 sectors/track with 256 bytes/sector }

Therefore:

vol_bytes = ((5 * 370) DIV 16) * 35 * 256
= 1 030 400 (bytes)

The tpm, bpt, and nvols parameters for 913x hard discs are found in the medium_parameters function in module ctr.

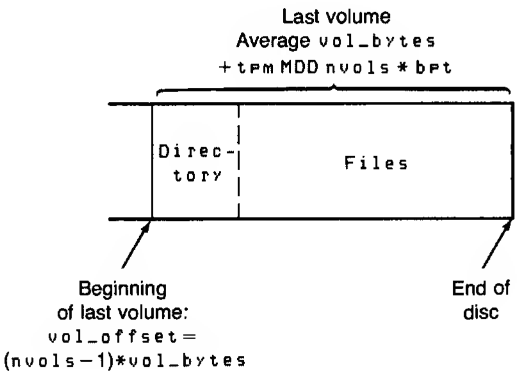
Here is a diagram of how the TABLE program partitions hard discs:



The first directory is placed at the "beginning" of the disc (at an offset of 0 bytes on track 0). The data area used for files immediately follows the directory.

The next directory is placed so as to just follow the end of the first volume. The size of the first volume determines the actual location where the second logical volume will begin. This rule is also followed by each successive logical volume on the disc.

The last volume on the disc looks as follows:



When TABLE attempts to validate unit numbers, it looks at these logical volume boundaries in search of valid directories. As each valid directory is found, the corresponding volume is assigned a unit number. If a valid directory is not found at its expected location, then the corresponding unit is invalid; the amount of disc space normally occupied by this volume can be coalesced with the last preceding logical volume found to have a valid directory, if desired.

Partitioning Recommendations

Here are the general recommendations as to how you can change the standard partitioning of hard discs:

1. The simplest method of changing the partitioning on your hard discs is to "coalesce" adjacent logical volumes. You should try to use this solution if possible.
2. If coalescing does not provide you with an adequate solution, you can also set up your own logical volume structure on the disc by modifying the parameters in the CTABLE source program.
   a. The easiest changes might be to change the `nvols` parameter in the `templates` section that corresponds to your disc. For instance, changing this constant from 30 to 15 for the 7912 allows you to have two 7912 drives automatically assigned unit numbers by CTABLE. The size of the logical volumes will be doubled, and partitioning will still be made on track boundaries.
   b. If the above methods still do not provide an adequate solution, read the subsequent discussion in Designing Your Own Partitioning Schemes.

Coalescing adjacent hard disc volumes was discussed earlier. Modifying the TABLE Program is discussed momentarily.

Note

If you do modify the standard TABLE program, keep in mind that you must use a version of the program that uses the *same* partitioning scheme in order for all logical volumes to be recognized properly.

Designing Your Own Partitioning Schemes

The most highly recommended method is the standard TABLE partitioning method. Here is the section of the `template` (toward the end of the main CTABLE program) that performs the standard partitioning:

```
for i := 0 to nvols-1 do
  tea_CS80_mv(11+i, primary_dam, {sc} 7, {ba} 0, {du} 0, {dv} 0,
                    vol_offset(i, nvols, mp),
                    {devid} CS80id,
                    vol_bytes(i, nvols, mp));
```

The vol_offset and vol_bytes functions calculate the offset and size of each of your directories according to the nvols and mp values; you can use the standard values, provided in this same template, or specify your own. If, for example, you wanted to change only the value of nvols for a 7908 disc to 8, you could change this line in the template (just a few lines before the standard partitioning algorithm shown above):

```
CS80id := 7908; nvols := 16; mp.tpm := 5* 370; mp.bpt := 35*256; {7908};
```

to this:

```
CS80id := 7908; nvols :=  8; mp.tpm := 5* 370; mp.bpt := 35*256; {7908};
```

The vol_offset and vol_bytes functions would then make the volume offset and size calculations for you.

While the standard method is the most highly recommended, there is nothing that prevents you from using your own. If you like, you may remove the for statement, duplicate the tea procedure call n times, and specify volume offsets and sizes of your choosing for each logical volume. Here is an example of one for unit 11 (you will have to supply actual values of the example parameters offset_for_unit_11 and bytes_for_unit_11 shown below):

```
tea_CS80_mv(11, primary_dam, 7, 0, 0,
                   offset_for_unit_11,
                   CS80id,
                   bytes_for_unit_11);
```

The tea procedure checks to ensure that your logical volumes each lie inside the media boundaries. Unfortunately, the tea procedure doesn't check to see if any of them overlap!

In those templates capable of partitioning media, you will find the following line:

```
{ mp := block_boundaries(mp); {override track boundary partitioning}
```

This allows you to use the standard partitioning method, except that the partitioning will occur on 512-byte block boundaries – not necessarily on track boundaries. The "{" character at the beginning of the line makes the line a comment; enable compilation of the line by deleting the "{" character. Depending upon the media parameters and the number of logical volumes, this may or may not make a difference in how your media actually gets partitioned. This feature is provided solely for compatibility with discs used with Pascal 1.0. If you don't need it for this reason, don't use it!

All parameters in the templates have typical values for your convenience. If you get a "value range error" when you execute your modified version of CTABLE, it probably means that one or more of your parameters is out of range. Don't worry about your system configuration; the old configuration will still be in effect. You can immediately go back to the Editor to try to determine the problem with your new CTABLE.

To find where the value range error occurred, usually the quickest way is to examine the tea procedure calls you just modified, and then examine the tea procedure itself to see what range it checks the parameters for. However, unless you are a certified wizard, don't modify the tea procedure itself!

If you still can't find the source of the error, you can re-compile CTABLE with $DEBUG ON$. Get a listing from the Compiler, too. Then execute CTABLE again. When it terminates with the error again, use the queue (Q) command in the Debugger to determine the line numbers of the statements leading up to the error. Also, when you examine the queue, you may need to trace back several line numbers to actually locate the offending statement.

System Unit Auto-Search Declarations

These constants determine the order of devices searched while trying to find a system volume.

```
{system unit auto-search declarations}
  const
    sysunit_list_length =
      7;
  type
    sysunit_list_type =
      array [1..sysunit_list_length] of unitnum;
  const
    sysunit_list =
      sysunit_list_type[
      harddisc_first_lun, {first hard disc logical unit number}
      45,   {srm, prefixed to user's sysvol}
      4,    {floppy unit 1, primary dam}
      44,   {floppy unit 1, secondary dam}
      3,    {floppy unit 0, primary dam}
      43,   {floppy unit 0, secondary dam}
      42];  {bubbles}
```

If a valid directory is not found on any of these units, then the system volume is determined by the normal algorithm (described in The System Volume section of The Booting Process discussion presented earlier this chapter).

If a system unit was expilcitly specified by modifying the constant called `specified_system_unit` at the beginning of the module called `options`, then this search will not override the specified system unit.

HP-IB Select Codes Searched

These constants determine the select codes scanned in search of an HP-IB type interface, including 98625 High Speed Disc interfaces.

```
{HP-IB select code scanning declarations}
  const
    sc_list_length =
      3;
  type
    sc_list_type =
      array [1..sc_list_length] of shortint;
  const
    sc_list =
      sc_list_type [
        7,      {internal HP-IB}
        8,      {default sc for HP98624 HP-IB}
        14];    {default sc for HP98625 HP-IB}
```

The select codes are searched in the order they appear in the list (7 first). On each select code, addresses 0 through 7 are polled in succession for devices. In the case of multiple devices contending for an assignment class, say multiple local hard discs where the total capacity of all is greater than 30 Mbytes, generally the last one polled will be the one assigned a logical unit number.

About Module CTR

This module should not be modified!

Built into it is a lot of knowledge about the supported HP mass storage products, and provides a general structure into which can be inserted information about new peripherals as they are introduced.

Each peripheral is assigned a letter designator; these are listed in the export section of module `ctr`. In addition there is descriptive information about the size of each type of device, expressed in bytes per track and tracks per medium. The routines in `ctr` avoid partitioning across track boundaries, which would cause very inefficient disc access patterns.

Most of the procedures exported from `ctr` are given a name prefixed with `tea_` . These are the Table Entry Assignment routines. There are `tea` routines for all the supported mass storage products. Some `tea` routines are appropriate for an entire family of related mass-storage products.

There are also some utility routines. The `create_temp_unitable` procedure allocates in the heap a temporary structure like the real system Unit Table. CTABLE makes its assignments to this temporary structure, then uses `assign_temp_unitable` to copy the final result into the actual system table. Note that `assign_temp_unitable` will not overwrite any RAM volumes which have been created in the system unit table. This feature is provided so that if you execute a CTABLE while the system is running, you won't lose files in memory.

The `sysunit_ok` function checks to see if a particular unit is blocked, on-line, and has a valid directory; if so, it is a legal candidate for the system unit.

If you look at the assignments to the various fields of a Unit Table entry, you will be aware that two of them are procedure variables which must be initialized to the names of the DAM (Directory Access Method) and TM (Transfer Method or driver) appropriate to the volume and physical device. DAMs and TMs are not part of CTABLE and so would ordinarily be linked to modules already in RAM by the linking loader when CTABLE is loaded.

However, there is no guarantee that the DAMs and TMs for a device are present, since they may have been removed from INITLIB or never even installed. Consequently, CTABLE has been programmed to examine the symbol tables kept in memory by the linking loader. If a driver's name is found, it can be used; otherwise, the program avoids references to absent drivers. The routine which searches for link symbols at run-time is called `value` and is exported from module `ctr`.

About Module BRSTUFF

This is another module which shouldn't be modified!

It exports two routines. The function `internal_mini_present` determines if there are any internal flexible disc drives in your computer. The function `get_bootdevice_parms` determines what type of device was used for booting and returns the `dav` (device address vector) for that device.

About Module SCANSTUFF

This module shouldn't be modified!

Its purpose is to interrogate certain disc drives about their size and identification. To do this, the `value` routine (see module `ctr`) is used to find routines which are present only if the driver modules supporting these discs are installed.

Discussion of the Main Body of CTABLE

A lot of details of the behavior of CTABLE can be modified by changing declarations such as the select code list from the options module. If you want to force some particular assignment, this may be achieved by modifications to the code in the body of CTABLE.

Default DAV Assignments

The program first assigns default device address vectors (DAVs) for devices that cannot be found by scanning (such as printers and HP 9885 8-inch disc drives).

HP-IB Interfaces Scanned

After various initializations, CTABLE scans the select codes listed in module options. For each HP-IB interface found, and for bus addresses 0 through 7 on each interface, the program inquires to see if a device is present. A letter designating the device is returned. You can see the definitions of these letters in the constant declaration at the beginning of the `ctr` module.

Boot Device Info

The information about the boot device is obtained. This may be used later in selecting the system unit.

Temporary Unit Table

A temporary Unit Table is then created in the heap. The assignments made as CTABLE executes will be made to elements of this temporary table; only at the end will the real system Unit Table be updated.

Standard Assignments

Next, "standard" unit number assignments are made. It is wise not to change these assignments, since programs tend to depend on them.

- Unit #1 is assigned to the screen (CONSOLE:)
- Unit #2 is assigned to the keyboard (SYSTERM:)
- Units #3 and #4 will be assigned to the highest priority flexible disc drive. If both internal drive(s) and an external flexible disc drive are present, the internal drive(s) will be used for #3 and #4 unless the external disc was the boot device. This policy gives preference to the higher-performance internal floppy disc drives.
- If an SRM interface is present, it is assigned unit #5. (It may also be assigned unit #45 later in the program.)
- Unit #6 is assigned to the local printer (PRINTER:). This assignment is made whether or not a printer is actually connected to the computer, because there is no way to interrogate every possible type of printer.

Additional Floppy Unit Pairs

Next, the second and third pair of flexible disc drives are assigned unit numbers. Units 7 and 8 are assigned to the second highest priority floppy drive pair, and 9 and 10 to the third priority pair.

Multiple Local Hard Discs

With auto-configuration, CTABLE can deal with several local hard discs found during the HP-IB scanning process (previous versions of this program, without modification, could only find one). This code is surrounded by conditional Compiler options, because you may wish to not compile it and instead force particular assignments.

CTABLE will break a hard disc (which has not previously been initialized to a single volume) into multiple volumes. As things are arranged (see module `options`), no volume will be less than one million bytes and no disc will be divided into more than 30 volumes. The units assigned to these volumes begin with #11 and can use up through #40, depending on the number required for each disc.

DC600 Tape Drives

If there are any DC600 stand-alone tape drives present, or any CS80 disc drives with integrated tape drives present, then the program also finds them. The highest priority tape is assigned unit number 41, and the second priority tape is assigned unit number 42.

Alternate DAMs

Next, the alternate-DAM entries are assigned. This allows all flexible discs to be used regardless of the resident directory type. Units #43: and #44: are alternates for #3: and #4:. For instance, if LIF is the primary DAM, then units #43: and #44: will use the alternate UCSD DAM to access units #3: and #4:. (Alternates for units #7:-#10: are a few lines later in the program.)

Duplicate SRM Unit Entries

The "duplicate entries for prefixing down the SRM" section provides templates that you can use to assign additional unit numbers to SRM directories. For instance, suppose you want to have unit #46: assigned to the directory called `/SPECIAL/USER10/FRED`. Enable the first template by deleting the `{` comment brace preceding it. Then scroll down until you find the comment `{ prefix the primary and secondary SRM unit entries }`. (It may be easier to use the Editor's Find command, since these templates are a couple of pages away from the first templates.) Enable the template for #46: by deleting the `{` comment brace, and replace the `?` with the desired directory path `SPECIAL/USER10/FRED`.

#45 is not really an alternate; it is another SRM volume, and may be assigned as the system volume later. If this happens, the operating system will have two units on the SRM: one for the "system volume," which is used for temporary system files, work files, stream files etc.; and another for the "default" working directory. This avoids any possible need to prefix an SRM system volume away from an SRM default volume.

More Alternate DAMs

Next, units #47: and #48: are assigned as alternate DAM units for #7: and #8: (second priority floppy disc pair). Units #49 and #50 are alternates for #9: and #10: (third priority floppy disc pair).

Templates

Next are the "templates" for overriding the mass storage table entry assignments made by the standard TABLE. These templates are surrounded by conditional `$if false$` Compiler options which cause them to be skipped. Thus, the `tea` procedure calls have no effect until you change the `$if false$` to `$if true$`. The `tea` procedures themselves, are defined in the module `ctr`. They actually perform the Table Entry Assignments.

There are templates for the following disc drives: internal; 8290x (Amigo); 9895; 913xA, B, V, and XV; SS80 flexible discs (such as the 9122) and CS80 hard discs (HP 7908, 7911, 7912, 7914, 7933, and 7935); CS80 tapes; EPROM cards; Magnetic Bubble memory cards. Each template gives the opportunity to specify the following:

- directory access method (DAM)
- select code
- bus address (HP-IB interfaces)
- drive unit
- offset in bytes from beginning of volume to this unit's directory (for multi-volume discs)
- drive type (the variable named letter in a constant declaration of module `ctr`)
- size of volume

For multiple-volume drives, the templates include a `for` loop which calculates how to break up the disc space in the preferred fashion.

If you want to change the default for an HP 9121 drive, you will need to use the HP8290X `tea` procedure. The reason for this is that the HP 9121 drives behave just like the HP 8290X drives. You might also note that you would also use the HP8290X `tea` procedures for the 5.25-inch drive in the HP 9135 and the 3.5-inch drive in the HP 9133.

The first parameter in the `tea` procedures specifies the unit number you wish to assign. It must be in the range from 1 thru 50. The second parameter specifies the directory access method, or DAM. The DAM specifier is of emumerated type "ds_type". Exported from module `ctr`, ds_type is shown here.

```
type
  ds_type =  {Directory access method Specifier for local mass storage}
    ( primary_dam,      {either LIF or UCSD, as specified in options}
      secondary_dam,    {the one not selected as primary}
      LIF_dam,          {LIF, regardless of primary/secondary choice}
      UCSD_dam   );     {UCSD, regardless of primary/secondary choice}
```

A `tea` procedure has parameters only for those items which are applicable to the device. Furthermore, all parameters are range-checked by the `tea` procedure. While the range-checking cannot guarantee the correctness of your parameters, it can nearly guarantee that your parameters won't ruin the system.

The remaining parameters for all the local mass storage `tea` procedures are device-specific. Most devices will need addressing information such as select code (`sc`), HP-IB bus address (`ba`), and disc unit number (`du`).

You may leave the templates where they are, or you may move them. However, all `tea` procedure calls must take place between these two statements:

```
{ Create a temporary table & fill it with dummy entries }
create_temp_unitable;
```

Place all `tea` procedure calls here.

```
{ assign the new unitable and unitclear all units }
assign_temp_unitable;
```

You may assign and re-assign logical units as many times as desired between the two statements above. When the same logical unit is assigned multiple times, the last assignment performed will be the one that remains in effect.

Temporary Unit Table Copied

Next, the temporary unit table is copied into the system's unit table (except that RAM volume entries are not overwritten).

SRM Prefix Directories

The SRM unit entries are then prefixed to the appropriate directories. Each workstation in an SRM system has an identification number called its "node number", and it is **strongly** recommended that the system be configured so that every workstation's node number is unique.

CTABLE tries to prefix #45 to a directory called /WORKSTATIONS/SYSTEMnn, where nn is the node number. If no such directory exists, it tries to use directory /WORKSTATIONS/SYSTEM (with no node number). If that one doesn't exist, entry #45 is nullified. This is a rather key mechanism. It allows the workstations in an SRM system to have unique configurations. For the normal functioning of the Pascal system, a system volume is required to hold the system library and various system files. If all workstations shared the same system volume, file name collisions would be a real nuisance. CTABLE supports this partitioning, and so does the overall booting process, allowing for instance a different INITLIB and TABLE for each workstation.

Remove Extraneous Hard Disc Volumes

When a valid directory is not found at the expected location on the disc, then the corresponding unit number is not valid. This service is performed by the section of code in the main part of CTABLE that follows the comment:

```
{ remove extraneous local hard disc entries if necessary }.
```

If desired, the volumes which don't have valid directories may be "coalesced" with the last valid directory found which precedes this invalid directory.

System Unit Selected

The system unit is then selected according to the priorities set in the constant called `sysunit_list`, exported from module `options`.

SRM System Unit Selected

Finally, if the system unit is #45: (SRM system volume), then unit #5 is also an SRM volume. In that case, #5 is prefixed back to the root SRM directory (`#5:/`) so the root is the initial default volume for the system right after it boots up. You can change the working directory to your own directory by adding the directory path to the slash (`/`).

System Files Re-Opened

This procedure re-opens the standard unblocked system 'files':

- #1: is assigned to SYSTERM:
- #2: is assigned to CONSOLE:
- #6: is assigned to PRINTER:

Editing CTABLE

If you have just read through the preceding discussion the first time, you will need to go back and read the relevant sections and make the desired changes.

If you have already edited the CTABLE source program, you are ready to store your new file. Quit editing and Write the edited CTABLE in a new file, such as NEWCTABLE (or use the Save option if you are using a backup copy of the file). Exit the Editor by typing E.

Compiling and Running CTABLE

1. The modules in CTABLE.TEXT import modules from INITLIB. However, the interface text for these modules is not available unless you enable the `$search 'CONFIG:INTERFACE'$` Compiler option at the beginning of the source program (by removing the comments from the line). You must also be sure that this disc is on-line during the compilation of the CTABLE program; you could also copy the file onto another on-line disc and change the volume specification in the program accordingly.
2. Load the Compiler by typing C (you may need to put the CMP: disc on-line). Answer the Compiler's `Compile what text ?` prompt by entering:

   ```
   NEWCTABLE
   ```

3. Answer the "Printer listing ?" prompt with:

   Y for a listing.
   N for no listing.
   E for an "errors only" listing (if you have a printer).
   L for a listing file.

4. Press Return or ENTER to say that the default output file name of "SYSVOL:NEWCTABLE.CODE" is fine.

   If you followed the example, you shouldn't have any compilation errors.

5. Press R or RUN to execute the new CTABLE.

Verifying the New Configuration

Generally, the Filer provides the quickest way to verify your configuration. The Volumes command provides a quick sweep of all units. The List command provides a way to test individual units.

Remember that the Volumes command shows only those units which are on-line and which have valid directories. It won't show units with media containing either no directory or the wrong type of directory.

If the first attempt to List the directory of a unit fails, the Filer displays:

```
Please mount unit #9
'C' coninues, <sh_exc> aborts
```

Type C. The Filer will then give the reason for failure. A key result is "no directory on volume", which means that the device and medium are accessible, but no directory was found. Other results such as "device absent or unaccessible", "medium absent", or "device not ready" mean that the attempt to read from the device failed.

If you get "device absent or unaccessible", there may be several possible reasons. A good trick at this point is to eXecute ACCESS:MEDIAINIT on the unit number of interest. For those device types MEDIAINIT recognizes, it will print out the expected device type, plus the addressing information. This is an excellent way to verify the expected configuration, even if the device itself is unaccessible. Don't worry about specifying a device that you really don't want to initialize; MEDIAINIT always prompts for your confirmation before it begins initializing.

Making the New Configuration Permanent

Once you are satisfied with your new configuration and wish to make it permanent (i.e., it will remain unless you change it again), copy the code file to your BOOT: disc. First, however, you should link the new file to itself in order to conserve disc space.

Link the Modules Together

1. Invoke the Librarian by inserting the ACCESS: disc and pressing [L].
2. Insert the SYSVOL: disc, press [I] (for Input) and enter:

   ```
   NEWCTABLE
   ```

3. To conserve space on the disc, you can specify a header size smaller than the default (38). Press [H], and enter: 1. The header size is then changed to the minimum (18).
4. Press [O] (for Output) and enter:

   ```
   NEWCTABLE
   ```

5. Press [L] (for Link).
6. Press [D] (to remove the file's Def table).
7. Press [A] (to link All the modules).
8. Press [L] (to finish Linking).
9. Press [K] (to Keep the file).
10. Press [Q] (for Quit).

Now you are ready to perform the final operations.

Install the New TABLE

1. Insert the ACCESS: disc and type F (for Filer).
2. Remove the original TABLE file. Insert the BOOT: disc, press R (for Remove) and enter:

   ```
   BOOT:TABLE
   ```

3. Krunch the BOOT: disc, since your new TABLE file may be larger than the old one. Press K (for Krunch) and enter:

   ```
   BOOT:
   ```

4. Respond to `Crunch directory BOOT: ? (Y/N)` with Y.
5. Now copy the new code file from SYSVOL: to BOOT:, giving it the required name. Insert the SYSVOL: disc, press F (for Filecopy) and enter:

   ```
   NEWCTABLE.CODE,BOOT:TABLE
   ```

6. Swap discs as directed by the Filer.
7. Save your new source file on the CONFIG: disc too. Insert the SYSVOL: disc, press F and enter:

   ```
   NEWCTABLE.TEXT,CONFIG:$
   ```

8. Swap discs as directed by the FILER.
9. Clean up the SYSVOL: disc by removing all the files you put there. Use wildcards to save typing. Insert the SYSVOL: disc, press R, and enter the ? wildcard.
10. Respond N to the prompt to remove LIBRARY, and respond Y to the prompts to remove INTERFACE, NEWCTABLE.TEXT, and NEWCTABLE.CODE. Respond Y to the confirmation prompt.
11. Exit the FILER by typing Q.

Example SRM Configuration

The Shared Resource Management (SRM) System is a "file server" system that allows several workstation computers to share file-oriented devices like discs, printers, and plotters. Also, the SRM may be the only mass storage device for a machine with no local disc drives.

This section explains how to configure workstations to access and boot Pascal 3.0 from an SRM system. It is used as the example "custom" configuration because it can employ three methods of modifying the standard configuration:

- Copying and re-naming files
- Adding modules to INITLIB
- Modifying the TABLE program (optional).

This section tells what to do the *first* time you set up the *first* Pascal workstation to access an SRM system. It should not be repeated for every workstation you set up. Once this procedure is complete, the SRM will be accessible any time you boot up your workstation.

Prerequisites

Here are the assumptions made by this set-up procedure.

Who Should Set Up the SRM

The person who is designated as the "SRM system administrator" should perform the process described in the next few pages.

SRM Hardware

It is assumed that your SRM hardware has been installed and tested as prescribed in the SRM documentation. In order for your system to work with the SRM, every workstation in the SRM configuration must have a unique node number (see the SRM System Manual to learn about node numbers). You will also need the wiring chart and node number assignments which were prepared when designing and installing your SRM system.

SRM 1.0 Operating System Parameters

There are four parameters which are set when the SRM 1.0 Operating System is initially configured. (These are only needed if you are using version 1.0 of the *SRM Operating System*; with SRM 2.0, they are set automatically.) Appropriate values for these parameters when using Pascal Workstations with the SRM 1.0 are as follows:

- IOBUFFERS: At least five per workstation in the cluster – for example, 40 buffers for 8 workstations.
- DISC BUFFERS: Fifty is a good choice.
- TASKS: Two is enough.
- FILES: Allow for ten or twelve open files per workstation in the cluster; one hundred is a nice round number.

Boot ROM Versions

If you have an HP 9816 Computer with a Boot ROM 3.0L, then you must boot from a local disc drive. The SRM can only be used after normal booting is complete. Similarly, if you have an HP 9826 or 9836 Computer with a Boot ROM with version number less than 3.0, then you must boot from the internal 5.25-inch flexible disc drive. In both of these cases, you will probably want to make a back-up copy of the original BOOT: disc, as you will be modifying the INITLIB file on that disc.

If your computer is equipped with Boot ROM 3.0 or later version, it is possible to boot directly from the SRM. System Boot files are found on the SRM system in the /SYSTEMS root directory; they have names like SYSTEM_P. The other files used at boot time (INITLIB, STARTUP, and TABLE) are found in the /WORKSTATIONS/SYSTEM directory. This too is explained in the SRM System Manual.

Overview of SRM Installation

Configuring your system to access SRM is not a hard or complicated operation, but it is important that you follow the subsequent procedures in exact detail. Since you are less likely to make mistakes if you understand what's going on, here is an outline of what you will do.

1. Install driver modules DATA_COMM and SRM by executing them (they are actually programs that install themselves automatically).
2. Execute the TABLE auto-configuration program. When it is executed while the DATA_COMM and SRM driver modules are installed, it will find the SRM system and assign unit #5 to the SRM.
3. If they are not already on the SRM system, create directories /SYSTEMS and /WORKSTATIONS/SYSTEM.
4. Copy the system Boot file (SYSTEM_P) to the /SYSTEMS directory. Copy the rest of the Pascal system files to the /WORKSTATIONS/SYSTEM directory. (The Boot ROM expects to find the Pascal system in these directories.)
5. Use the Librarian to create (on the SRM) a new INITLIB file that contains modules DATA_COMM and SRM, and then replace the existing INITLIB with this new one. (If you have Boot ROM 3.0 or later, then you will be replacing the INITLIB in the /WORKSTATIONS/SYSTEM directory; with earlier Boot ROMs and Boot ROM 3.0L, you will be replacing the INITLIB on the BOOT: disc.)
6. Re-boot the computer, and verify the new configuration.
7. You can also optionally modify the TABLE program to assign additional unit numbers to the SRM system.

Installing the SRM Driver Modules

First, install module DATA_COMM. The file is on the CONFIG: disc that is supplied with your system. Although you may have already copied the file onto another volume, such as a local hard disc, this example assumes that you will be loading and executing it from the CONFIG: disc.

Execute the file by pressing X at the Main Command Level. The system prompts:

```
Execute what file?
```

Enter this file specification:

```
CONFIG:DATA_COMM.
```

Be sure to include the trailing period to suppress the ".CODE" suffix.

Install the SRM module similarly; it is also on the CONFIG: disc as shipped to you.

Re-Configuring with TABLE

Use the eXecute command to execute the TABLE program; it is on the BOOT: disc supplied with your system. Press X, and then answer the `Execute what file?` prompt with this file specification:

```
BOOT:TABLE.
```

Again, be sure to include the trailing period.

When the program has finished, you can use the Filer's Volumes command to see that unit #5 is assigned to the SRM system. From the Main Command Level, press F and then V. Here is a typical display:

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 # BOOT:
  5 # ROOT1:
  6   PRINTER:
```

If the name of the SRM's root directory is not shown in the display, re-execute all three programs (DATA_COMM, SRM, and TABLE). You may have done something wrong in that process.

If the Filer's Volumes command still does not recognize the #5: volume, check to see whether the SRM hardware is properly configured and installed. For instance, the (unmodified) TABLE program expects that the SRM interface in your computer is set to select code 21.

If that does not work, then you should refer to the troubleshooting sections of the SRM System Manual.

Creating the Required Directories and Files

The first time that a workstation is set-up to access an SRM system, you will need to set up certain directories on the SRM. These directories have special functions, as described in the following paragraphs.

A Sketch of Normal SRM Directory Configuration

In order to allow each Workstation in an SRM configuration to boot up a unique system and have its own system volume, a private directory is established for each node number.

Strictly speaking, this is not always necessary. If a workstation has a local high-performance mass storage device, then it may be desirable to use that device as the system volume. In fact, the automatic configuration process will select a high-performance mass storage as the system volume, if one is present. However, it doesn't hurt anything to set up unique directories for each workstation. The following discussion explains how to do so. If things are first set up as explained below, you then have the option to copy frequently used files such as the Editor and Compiler from the SRM onto your local high-performance system volume. Then when you boot the system, those files will be found locally and accessed with correspondingly greater speed.

In the SRM's root directory there should be another directory called WORKSTATIONS. Under this there should be a directory called SYSTEM, and for each node number "nn" there should also be a directory called SYSTEMnn. For instance, if there are three Workstations on nodes 08, 14, and 15, then the following directories should exist in the root:

```
WORKSTATIONS/SYSTEM
WORKSTATIONS/SYSTEM08
WORKSTATIONS/SYSTEM14
WORKSTATIONS/SYSTEM15
```

Under WORKSTATIONS/SYSTEM should be copies of all the system files, such as the Compiler, Filer, and Editor.

Under the private directory for each node should be accessible all the files normally used by the Workstation. For files which don't change, such as the Compiler, it is sufficient to simply have a duplicate link to WORKSTATIONS/SYSTEM/COMPILER; there is no need to actually copy such invariant files. The Filer's Duplicate link command can be used for this purpose.

Also in a node's private system directory can be the files which "personalize" a Workstation: customized copies of LIBRARY, INITLIB, AUTOSTART, and so forth.

Once this set-up is created, booting is a smooth and automatic process. With Boot ROM 3.0 and later versions (but not 3.0L), you can boot from the SRM; the particular system to be booted is selected by name at power-up. Thereafter, the Workstation looks for the necessary files in the directory with its node number. If INITLIB can't be found in the /WORKSTATIONS/SYSTEMnn directory, default is taken to /WORKSTATIONS/SYSTEM; if something crucial is still missing, the boot may fail. (The computer will complain to the operator.)

If you boot from the SRM or if you have no local hard disc on-line, your system volume will be unit #45 (prefixed to your private directory /WORKSTATIONS/SYSTEMnn) and your default volume will be #5 (another SRM volume, prefixed to the SRM root directory). Even if the SRM is not chosen as your system volume (using the scheme above), it will still be accessible through units #5 and #45.

In order to run properly, there must be one more special directory called TEMP_FILES under /WORKSTATIONS. All temporary files are created in this directory, and are removed when no longer needed. If you don't create this directory, the first workstation to need it will do so. Should the create fail, an error is reported. Consequently the directory /WORKSTATIONS should not be write-protected unless directory TEMP_FILES has already been created.

Most users will also want a private directory for their default volume. Typically one creates a directory called USERS under the root, and within USERS a private directory for each individual. After booting, use the Filer to set the current working directory for your unit #5 to your private directory (you can modify the TABLE program or create an AUTOSTART file to do this for you). This keeps the root directory from getting cluttered.

Setting Up SRM Directories

Insert the ACCESS: disc in drive #3 and press F to execute the Filer. When the Filer prompt appears, press V to list the volumes on-line.

If the SRM has already been running with some other systems connected, such as an HP 9845 or 9836 running BASIC, some of these directories may already exist. To see the directories which already exist, press L for the List directory command, and enter the root-level directory specification:

```
#5:/
```

In following the steps below, obviously you should skip the steps which create directories which already exist on your SRM.

To create directory /WORKSTATIONS, use the following Filer sequence.

1. Press M for the Make-directory command. The Filer responds with this prompt:

   ```
   Make file or directory (F/D) ?
   ```

2. You want to make a directory, so type D. The Filer responds with this prompt:

   ```
   Make directory (valid only for SRM type units)
   Make what directory?
   ```

3. You enter this response:

   ```
   #5:/WORKSTATIONS
   ```

   Be sure to type this name in capital letters! If the root directory was protected with one or more passwords, the Filer would report: 'Error: invalid password' at this point. In such a case, you need to find out the required passwords from whoever initialized the SRM disc or installed the passwords. To create this directory, you need Write access rights in the root directory, and possibly Manager rights if they were specified. For instance, if the password for Write access is PLEASE, you would specify:

   ```
   #5:/.<PLEASE>/WORKSTATIONS
   ```

   Alternatively, you might use the main volume password by specifying:

   ```
   #5<VOL_PASS>:/WORKSTATIONS
   ```

   The Filer should reply:

   ```
   Directory is 'WORKSTATIONS' correct ? (Y/N)
   ```

4. You answer Y. The directory is created, then the Filer announces:

   ```
   Directory WORKSTATIONS made.
   ```

If the computers in the SRM configuration have Boot ROM 3.0 (or later version) which is able to boot from the SRM, you will also want to create a directory called SYSTEMS in the root. Repeat the steps just given, but instead specify that you want to create directory #5:/SYSTEMS.

Next, create directory SYSTEM under /WORKSTATIONS. This is where the master copy of all system programs such as the Compiler will be stored. To reduce the amount of typing involved, we will make the current working directory for unit #5 be the newly created /WORKSTATIONS directory.

5. Type P for the Prefix command. The Filer responds:

   ```
   Prefix to what directory ?
   ```

6. Enter:

   ```
   #5:/WORKSTATIONS
   ```

   The Filer will respond:

   ```
   Prefix is WORKSTATIONS:
   ```

Now if you don't specify a unit number in Filer operations, the system will assume you are referring to directory /WORKSTATIONS. To create SYSTEM, the sequence is as follows:

1. Press M
2. `Make file or directory (F/D) ? D`
3. `Make what directory? SYSTEM`
4. `Directory is 'SYSTEM' correct? (Y/N) Y`
5. `Directory SYSTEM made`

Also under /WORKSTATIONS create directories called SYSTEMnn, where nn is the node number for each workstation in the system. You can see why we said each node number should be unique! For example, create SYSTEM05 for the workstation at node 5. Note that two digits are always required, even if the first digit is zero.

Finally, under /WORKSTATIONS you should create a directory called TEMP_FILES. This is only necessary if you plan to write-protect /WORKSTATIONS.

Copying the System Files to SRM

You are now at the last stage! It is time to move the required files out into the new directories.

1. First prefix the current working directory to SYSTEM. Press P for the Prefix command.
2. Enter this directory specification:

   ```
   #5:/WORKSTATIONS/SYSTEM
   ```

   The Filer responds with this message:

   ```
   Prefix is SYSTEM:
   ```

3. Then insert the BOOT: disc in the drive you have been using and copy all the files on it into the new working directory. Press F for the Filecopy command. The Filer gives this prompt:

   ```
   Filecopy what file?
   ```

4. Specify that you want all files on the BOOT: disc to be copied by using the = wildcard as follows:

   ```
   BOOT:=,$
   ```

   The Filer will copy the files one after another.

Then repeat the above operation for each of the Pascal system discs (ACCESS:, SYSVOL:, etc). After this is done, the /WORKSTATION/SYSTEM directory contains the entire Pascal Workstation system.

Duplicating Links to System Files

Now you need to make these files available in the private SYSTEMnn directory of each workstation. For each such system directory, use the Filer's Duplicate Link command.

1. Press D.

   ```
   Duplicate link (valid only for SRM type units)
   Duplicate or Move ? (D/M)
   ```

2. You want to duplicate links rather than move links. Press D. The Filer will ask:

   ```
   Dup_link what file?
   ```

3. Answer:

   ```
   ?,#5:/WORKSTATIONS/SYSTEMnn/$
   ```

   Of course you should substitute a two-digit node number for nn each time (a leading 0 is required for single-digit node numbers). The "?" wildcard tells the Filer to ask if you want links each file in the source directory. Answer Y for every file except AUTOSTART and SYSTEM_P.

The Dup_link operation is very fast. It displays each file name as the links are made.

The last detail is optional. If any of the workstations in the SRM system have Boot ROM revisions 3.0 or later and will be expected to boot from the SRM instead of using local mass storage, you need to put a copy of the system Boot file in directory /SYSTEMS (not in /WORKSTATIONS/SYSTEM). The system Boot file (SYSTEM_P) is on the BOOT: disc shipped with the system; you probably have already made a copy of it in an earlier procedure. The Dup_link command can duplicate the file in a different directory.

1. Type D for the Duplicate link command.

   The Filer responds with this prompt:

   ```
   Duplicate link (valid only for SRM type units)
   Duplicate or Move ? (D/M)
   ```

   Respond with D.

2. The Filer prompts with this question:

   ```
   Dup_link what file?
   ```

   Respond with:

   ```
   #5:/WORKSTATIONS/SYSTEM/SYSTEM_P,#5:/SYSTEMS/$ Return
   ```

That concludes the required SRM software setup. Now any workstation using the BOOT disc you have created will be able to access the SRM via logical units #5 and #45. If a workstation has high performance local mass storage such as a fixed disc, that workstation's system volume will be on the local mass storage; otherwise the SRM directory #45:/WORKSTATIONS/SYSTEMnn will be the the system volume.

It is advisable to also create a private working SRM directory for each user, in addition to the SYSTEMnn directories for each workstation. Typically a user will then use unit #45 for his system volume and #5 will be prefixed to his working directory. A good way to set this up is to create a directory such as the following one in the root directory:

```
USERS
```

Then you can add subordinate directories like the following for each user:

```
USERS/TOM
USERS/DICK
USERS/HARRIET
```

SRM as the System Volume

At this point, you can make the /WORKSTATIONS/SYSTEMnn the system volume. You will first need to re-execute the TABLE program in order for unit #45: to be assigned to this directory. Press X at the Main Command Level, and enter this file specification:

```
/WORKSTATIONS/SYSTEMnn/TABLE.
```

Of course you will need to replace the nn with the node number of your workstation. Don't forget the period.

Now you can execute the Newsysvol command (at the Main Command Level) and specify `#45:` as the unit number. Then use the What command to verify that all of the subsystems (EDITOR, FILER, etc.) were found in the /WORKSTATIONS/SYSTEMnn directory. Changing the system volume will allow you to access the SRM copies of these subsystems by pressing keys such as E for Editor, and so forth.

Note

You should not prefix the working directory of unit #45: away from this directory.

Adding Modules to INITLIB

Now we will add modules DATA_COMM and SRM (on the CONFIG: disc) to INITLIB (on the BOOT: disc); actually, you will make a new INITLIB on the SRM that includes the drivers required for the SRM.

1. At the Main Command Level, press L to load the Librarian (note that the Librarian should be loaded from the SRM).
2. When you see the Librarian's prompt line at the top of the CRT, press O to specify the name of the (Output) file the Librarian will be creating.
3. Enter this file specification:

   ```
   #5:/WORKSTATIONS/SYSTEM/INITNEW
   ```

4. Press I so you can specify an Input file, then enter:

   ```
   #5:/WORKSTATIONS/SYSTEM/INITLIB.
   ```

   Be sure to type the period after the word INITLIB in this command (to suppress the otherwise automatic .CODE suffix). The Librarian will respond by showing INITLIB as the name of the input file.
5. Near the bottom of the CRT you will see a line which says:

   ```
   M input Module: KERNEL
   ```

   Press T to transfer this module to the output file. After a few moments, the name of a new module (KBD) will appear. Each time a new module name appears, press T to move it to the output file. You should continue copying modules until the name LAST appears; **Don't copy the module LAST yet.**
6. Now you must get the required SRM drivers and include them in the Output file. First close the Input file by typing an I and then entering a null response.
7. Press I for an Input file and enter this file specification:

   ```
   #5:/WORKSTATIONS/SYSTEM/DATA_COMM.
   ```

   Don't forget the period after the name.
8. When the module name DATA_COMM shows up near the bottom of the screen, press A which tells the Librarian to transfer all the modules in the file.
9. Then use the I command again to pick up the SRM input file, again being sure to type the period after the file name:

   ```
   #5:/WORKSTATIONS/SYSTEM/SRM.
   ```

10. Again transfer All by typing A.
11. Enter an I (Input file) command with null response. This closes the SRM file.
12. Press I for Input and enter the file specification of the original INITLIB file:

    ```
    #5:/WORKSTATIONS/SYSTEM/INITLIB.
    ```

13. When module KERNEL shows up near the bottom of the screen, select module LAST instead by pressing M for module and enter:

    ```
    LAST
    ```

    Then transfer it by typing T.
14. You now have all the modules in your new library. "Keep" it by typing K. Then quit the Librarian by typing Q.

Replacing INITLIB

Where you place the new version of INITLIB depends on which Boot ROM is in your machine.

- If you have Boot ROM 3.0 or later (but not 3.0L), then you will probably want to leave it in the /WORKSTATIONS/SYSTEM directory; it will be found there automatically when you boot from the SRM system.
- If you have an earlier Boot ROM or Boot ROM 3.0L, then you will need to replace the INITLIB on the BOOT: disc with the new INITLIB; this is required because these Boot ROMs cannot boot directly from SRM – they must use the BOOT: disc.

With Boot ROMs 3.0 and Later

1. Use the Filer's Change command to re-name the existing INITLIB (in /WORKSTATIONS/SYSTEM) to something like OLDINITLIB.
2. Use the Change command again to re-name the INITNEW file to INITLIB.
3. Re-boot your workstation to verify that the new INITLIB file works correctly.
4. Use the Filer's Dup-link command to link the new INITLIB to all /WORKSTATIONS/SYSTEMnn directories for the workstations that will be booting from the SRM. (You can alternately make custom INITLIB files for each workstation, if desired.)

With Earlier Boot ROMs

1. Press F to invoke the Filer.
2. Put in the spare copy of the BOOT: disc (*not* the original) into a drive. Press R for the Remove command. The computer responds with this prompt:

   ```
   Remove what file?
   ```
3. Answer:

   ```
   BOOT:INITLIB
   ```

   Note that there is no period after the file name this time.
4. Press K (Krunch) to pack all the remaining files on the disc to make the maximum amount of room for the new INITLIB. The Filer answers:

   ```
   Crunch what directory?
   ```
5. Answer:

   ```
   BOOT:
   ```

   Don't fail to type the colon after the volume name!

   The Filer will then say:

   ```
   Crunch directory BOOT ? (Y/N)
   ```
6. Answer Y. The computer then prompts:

   ```
   Crunch of directory BOOT in progress
   DO NOT DISTURB!!
   ```

Note

If you interfere with the disc before the Crunch operation completes, you will ruin the data on the disc. You will certainly have to recopy it from the original BOOT: and you may have to re-initialize it.

After the Krunch is complete, the filer prompts:

```
Crunch completed
```

7. Now when the Crunch is finished, you can Filecopy the INITNEW library file onto the new BOOT: disc. At the same time, you can re-name it INITLIB.

   Insert disc NEWLIB and press F for the Filecopy command.

   ```
   Filecopy what file?
   ```

   Answer:

   ```
   #5/WORKSTATIONS/SYSTEM/INITNEW.CODE,BOOT:INITLIB
   ```

   When the Filecopy finishes, you have a BOOT:INITLIB disc which contains the SRM drivers.
8. Verify that the new INITLIB works by re-booting your system.

Each Pascal workstation in the system with earlier (or 3.0L) Boot ROMs must boot using an INITLIB which has the SRM driver software installed. You may wish to make copies of the disc you've just created for each workstation. The disc can be copied using this Filer command sequence: F #3,#3. (You can alternately make custom INITLIB files for each workstation, if you want.)

Multi-Disc SRM

When an SRM system has more that one hard disc, you will need to modify, recompile, and execute the CTABLE program to allow access to these discs. This section describes how to perform this type of configuration change.

When more than one hard disc is installed on the SRM system, each disc must have a /WORKSTATIONS directory. If the directory is write-protected, then a /WORKSTATIONS/TEMP_FILES directory must be created. You may also wish to create another /SYSTEMS directory. Boot ROM 3.0 and later versions will search for bootable systems on each disc containing a /SYSTEMS directory.

CTABLE Modifications

Near the end of the CTABLE program, just above the manual templates section, a small section of code assigns Unit Table entries for the SRM.

```
with SRM_dav do
  begin
    {  tea_srm( 46, sc, ba, du);  {free}
    tea_srm( 45, sc, ba, du);  {for possible use as the system unit}
  end; {with}
```

The first tea entry provides a template for assigning unit #46: to the second hard disc connected to the SRM. You should change the du parameter to du+1 to specify the second disc.

Just below the manual "templates" section of the CTABLE program is another section pertaining to units for the SRM.

```
{ prefix the primary and secondary SRM unit entries }

if not unit_prefix_successful('#5:/') then {do nothing};
  {tries to set up uvid for possible default unit assignment below}

{ if not unit_prefix_successful('#46:/?') then zap_assigned_unit(46);  {free}

if not unit_prefix_successful('#45:'+srmsysprefix+srmnode(unitable^[45].sc)) then
  if not unit_prefix_successful('#45:'+srmsysprefix) then
    zap_assigned_unit(45);
```

If you remove the leading comment delimiter ({) from the #46: entry and remove the question mark from the literal '#46/?', then Pascal will be able to recognize the second hard disc connected to the SRM.

If you wish to have a Unit Table entry for a particular directory path name, you can include the path name in the specification. For example:

```
if not unit_prefix_successful('#46:/USER/AL') then zap_assigned_unit(46);
```

If you make this modification be sure to activate its accompanying `tea_srm` procedure by removing the curly brace.

```
    tea_srm( 46, sc, ba, du); {free}
```

With this modification, the system will boot with unit #46 assigned to the directory "/USER/AL" on the first SRM disc.

After all modifications have been made, you can compile CTABLE. Remember that you need to enable the `$search 'CONFIG:INTERFACE'$` Compiler option at the beginning of the program and make the INTERFACE library accessible at compile time. You will probably also want to link the resultant TABLE object file to itself with the Librarian to conserve disc space. See the procedures in the preceding section called Modifying the TABLE Program for explicit details.

Non-Disc Mass Storage

Chapter

10

Introduction

Pascal 3.0 supports several types of "non-disc" mass storage:

- Internal memory (RAM)
- HP 98259A Magnetic Bubble Memory cards
- HP 98255A EPROM (Erasable Programmable Read-Only Memory) cards
- DC600 Tape Drives (found in CS80-type disc drive units)

The Bubble and EPROM cards and tape drives provide non-volatile mass storage of programs and data; internal memory is volatile. All of them can be accessed through the File System. However, Pascal will not recognize either Bubble or EPROM cards until a few modifications are made.

This chapter describes configuring and accessing Bubbles, EPROMs, and tapes. Using internal memory for mass storage is covered in the *Pascal 3.0 User's Guide* and in the description of the Memvol command in the Overview chapter.

Summary of Configuration Modifications

In order for the File System to recognize either the Bubble Card or the EPROM card, you need to make the following configuration changes:

- Add the appropriate driver-module to INITLIB
- Modify the TABLE auto-configuration program. The source program (CTABLE) already contains the necessary templates; you only need to make a few simple changes to enable them.

Tape drives will be recognized without changes to INITLIB or the TABLE auto-configuration program.

Mass Storage Comparison

The operating characteristics for various mass storage devices are compared in the following table.

Characteristics	Storage Device				
	Mini-Floppy Discs	Bubble Cards	EPROM Cards	Memory Volumes	DC600 Tapes
Storage Capacity	270 336	131 072	131 072[1] or 262 144	variable[2]	16 000 000 or 67 000 000
Relative Access Speed	moderate	slow	fast	fast	slow
Read/Write Capability	yes	yes	no[3]	yes	yes
Usable as a Boot Device	yes	yes[4]	yes[5]	no	no
Removable Media	yes	no	no	no	yes
Multiple Volumes	no	no	yes[6]	no	yes
Data Integrity	poor	good	good	moderate[7]	good
Relative Cost	low	high	moderate	moderate	low

[1] Size depends on EPROM device type. Sixteen 2764-type devices provide 131 072 bytes while sixteen 27128-type devices provide 262 144 bytes.

[2] Size is limited by available memory.

[3] EPROMs can be read just like RAM memory but must be programmed (written) with the HP 98253 EPROM Programmer Card.

[4] The CTABLE program must be modified to allow this boot device to be the default system volume.

[5] This device can be allowed as a boot device; however, there are several restrictions that apply. See the discussion of Booting from EDISCS for further information.

[6] Multiple volumes can be programmed into one EPROM Card, on 16 Kbyte boundaries.

[7] RAM memory reliability is dependent on power-source stability.

Using Bubble Cards

This section provides all of the information you will need to configure and access Bubble memory cards from the File System.

Power Constraints

Due to the amount of power consumed by a Bubble card when data is being transferred, no more than two Bubble cards can operate at the same time without exceeding the capacity of the power supply in the existing Series 200 Computers. It is further recommended that only one Bubble card be operating at the same time as any other "high-power" card (such as the HP 98620 DMA card).

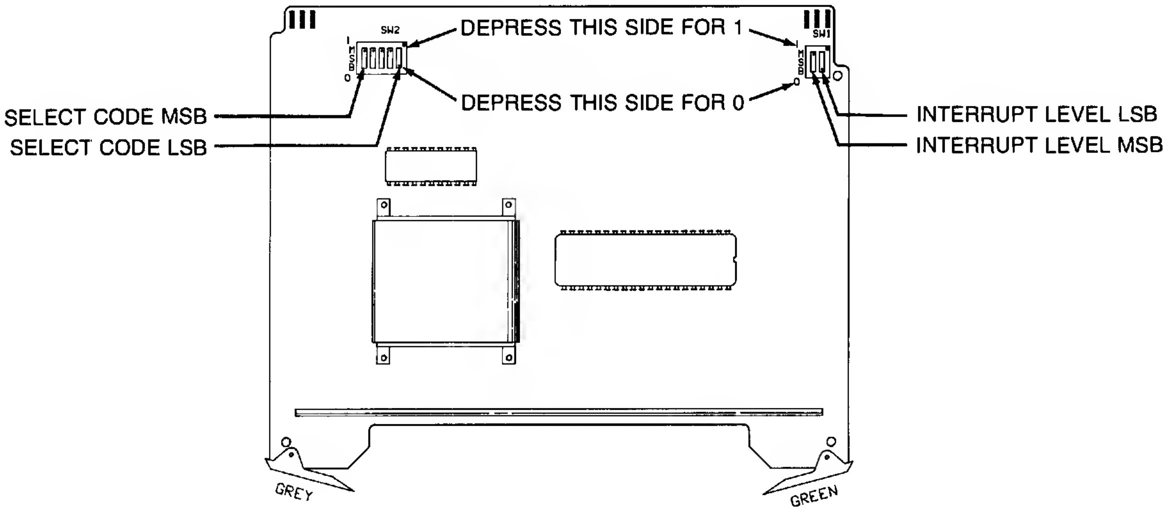
Bubble Card Configuration

If you have not already installed the Bubble card, see its installation note for complete details. Some of the installation information is repeated below for convenience.

CAUTION

ALWAYS TURN THE COMPUTER OFF BEFORE INSTALLING OR REMOVING INTERFACES.

The Bubble card has two banks of switches. The large switch bank sets the select code while the small one controls the interrupt priority.



Bubble Card Switch Locator

Select Code

The Bubble card's select code is preset at the factory to select code 30. If this select code conflicts with any another interface present in the system, change it to some *unused* value from 8 through 31. Note the select code setting; it will be needed for the changes to the TABLE program.

Note

If you change the select code of the Bubble card from its factory default setting, you must also change the CTABLE program accordingly.

Select Code Switch Settings

Switch Settings MSB 43210 LSB	Select Code	Switch Settings MSB 43210 LSB	Select Code
01000	8	10100	20
01001	9	10101	21
01010	10	10110	22
01011	11	10111	23
01100	12	11000	24
01101	13	11001	25
01110	14	11010	26
01111	15	11011	27
10000	16	11100	28
10001	17	11101	29
10010	18	**11110**	**30**
10011	19	11111	31

Interrupt Priority

The interrupt priority switches have been preset to level 5. Each Bubble card should be set to a *unique* interrupt priority since the Bubble card may lose data if interrupts are not serviced quickly. This is especially true if you plan to make calls directly to the driver procedure or use the overlapped I/O capability.

Interrupt Priority Switch Settings

Interrupt Level	Setting MSB LSB
3	0 0
4	0 1
5	1 0
6	1 1

If other interfaces have been installed which use interrupt level 5, change the switches on the Bubble card to the highest unused interrupt level in the range 3 through 6.

The Bubble card should now be ready to install in the computer. With the power turned *off*, install the card in the backplane. See the installation note if you have any difficulties.

INITLIB Driver Modules

The BUBBLE module is supplied on the Pascal 3.0 CONFIG: disc. The Pascal 3.0 IODECLARATIONS module recognizes Bubble cards as CARD_TYPE=8 (a field of the ISC_TABLE array in IODECLARATIONS). This same version also recognizes the EPROM cards.

Loading the BUBBLE Module

As with other driver modules, there are two ways to load the BUBBLE module:

- Execute it (with the Main Command Level eXecute command)
- Add it to INITLIB

Executing the module "permanently" loads the module, but must be performed every time the system is booted. Adding the module to INITLIB eliminates having to load the module each time and allows the Bubble card to be a candidate for use as the system volume.

Adding BUBBLE to INITLIB

If you have two disc drives, the creation of the new INITLIB is relatively simple. If you only have one disc drive, you will need to create a memory volume large enough to hold the new library (about 175K bytes).

To create a memory volume, press (M) from the Main Command Level. You will be prompted for the number of 512 byte blocks (answer 350) and the number of directory entries (answer 8). If you are not familiar with memory volumes, see the Memory volume command in the Main Command Reference section of the Overview chapter.

1. Initialize a disc, and then use the Filer's Filecopy command to copy the BOOT: disc onto this disc. Since you will be storing the new INITLIB on this new BOOT: disc, you can Remove the existing INITLIB file from the disc. Since the old INITLIB was probably not the last file on the disc (and the new INITLIB will probably be bigger than the old), you should Krunch the disc.

   (It is a very good practice to create a new BOOT: disc rather than modifying your present BOOT: disc. That way you can always return to where you are, no matter what happens to the new disc.)

2. Invoke the Librarian. This is done by pressing (L) from the Main Command Level. If the Librarian is not on-line, insert the ACCESS: disc and try again. Remove the ACCESS: disc once the Librarian has loaded.
3. Insert the *old* BOOT: disc into Unit #3 and the *new* BOOT: disc into Unit #4. (If you are using a memory volume, the memory volume will be the "blank disc". Use whatever unit number you assigned to the memory volume instead of Unit #4 for the remaining steps.)

4. Now use the Librarian to create the new INITLIB.Syntax:

   a. Press I and then enter the file specification by typing `#3:INITLIB.` and Return or ENTER. You must include a trailing period to prevent the Librarian from appending the `.CODE` suffix.

      When the Librarian finds the input file, the display will show the name of the first module in the file. You should see the module named KERNEL. If you have a printer, you can press F to list all of the modules in INITLIB.

      The BUBBLE module can be inserted anywhere after the IODECLARATIONS module but *before* the module named LAST. (LAST must be the last module in INITLIB.)

   b. Press O and enter `#4:BUBLIB.` as the Output file. Again, a trailing period prevents the `.CODE` suffix from being appended to the file name.

      (This disc must *not* be removed until you have finished creating the new BUBLIB file.) If you are using a memory volume, use the unit number of the memory volume.

   c. Press E to enter the Edit mode. You should now see this prompt (in the middle of the screen):

      ```
      F   First module: KERNEL
      U   Until module: (end of file)
      ```

   d. Press U and enter `LAST` as the Until module. You can now transfer all modules in the file up to (but not including) module LAST by pressing C.

   e. When the preceding transfer is complete, press A to append a module to the BUBLIB Output file. The Librarian prompts with `Input file:`. Put the CONFIG: disc, or whichever disc now contains the BUBBLE module, in Unit #3 (*not* #4, which must not be removed). Enter this file specification: `#3:BUBBLE.`.

   f. The Librarian now prompts with `Enter list of modules or = for all`. Enter `=`. After the BUBBLE module has been transferred to the BUBLIB library, the Librarian prompts with `Append done, <space> to continue`. If you removed the BOOT: disc (or the one that contains the INITLIB Input file) to put in the CONFIG: disc, replace the BOOT: disc now *before* pressing the spacebar to answer the prompt.

      (If you removed the BOOT: disc in #3: and did not replace it before pressing the spacebar, you get the following message:

      ```
      cannot open '#3:INITLIB', ioresult = 10.
      ```

      In such case, don't worry. Remove the CONFIG: disc and insert the BOOT: disc, then press I and enter `#3:INITLIB.` as the Input file. Press E to return to Edit mode, and return to where you were previously by pressing F and entering `LAST` as the First module. Proceed with step g below.)

   g. Press T to transfer module LAST to the BUBLIB file (if you got the error described in the preceding step, press C instead of T). Then press S to stop editing and K to keep the file.

   h. You should now verify that the BUBBLE module was indeed copied to the Output file. Press I and enter `#4:BUBLIB.` as the new Input file. Press the spacebar repeatedly to scan through the modules in the new library file. If you have a printer, press F to get a File Directory listing.

   i. If all modules are present, then press Q to Quit the Librarian.

5. If you have been using two discs, use the Filer to Change the file named BUBLIB (on the new BOOT: disc) to INITLIB. If you used a memory volume, remove the old BOOT: disc from Unit #3 and insert the new BOOT: disc; then use the Filer to Filecopy BUBLIB from the memory volume to the new disc, changing the file name to INITLIB in the process.
6. Re-boot the computer, which installs the new INITLIB containing the BUBBLE module.

Once the BUBBLE module has been installed, the Bubble card can be accessed by procedure calls. (The procedure calls will be discussed later.) To make the Bubble card available to the file system as a mass storage unit, the CTABLE program must be modified to reserve an entry in the Unit Table.

CTABLE Modifications

The Pascal 3.0 CTABLE program, supplied on the CONFIG: disc, contains a "template" for the Bubble card. You can either use the Editor's Find command to search through all occurrences of `BUBBLE` until you find the template, or Jump to the end of the program and scroll up until you see the BUBBLE template shown below.

```
 $if false$ { BUBBLE memory }
     {watch for conflicting uses of unit 42}
     {BUBBLE_DAV.SC default is 30 but may have been changed to boot SC}
     tea_BUBBLE(42,primary_dam,BUBBLE_dav.SC);
$end$
```

Change `$if false$` to `$if true$`.

```
$if true $ { BUBBLE memory }
    {watch for conflicting uses of unit 42}
    {BUBBLE_DAV.SC default is 30 but may have been changed to boot SC}
    tea_BUBBLE(42,primary_dam,BUBBLE_dav.SC);
$end$
```

This is the only modification that must be made to CTABLE for the system to recognize the Bubble card. It assigns unit number 42 to the Bubble card. If you are already using unit number 42, change the unit number to one that you are not using.

If you have more than one Bubble card or wish to have the Bubble card as a possible system volume, you should consider the following modifications.

Multiple Bubble Cards

If you install more than one Bubble card, a separate "tea" procedure must be executed for each card. An example is shown below.

```
$if true $ { BUBBLE memory }
    {watch for conflicting uses of unit 42}
    {BUBBLE_DAV.SC default is 30 but may have been changed to boot SC}
    tea_BUBBLE(42,primary_dam,BUBBLE_dav.SC);
    tea_BUBBLE(20,primary_dam,28);
    tea_BUBBLE(21,primary_dam,29);
    { tea_BUBBLE(3,primary_dam,30); {This would override #3}
$end$
```

For each tea_BUBBLE procedure called, you should specify an unused unit table entry, the type of directory access method (the LIF DAM is recommended), and the select code (switch setting) of the Bubble card. Since these templates override the auto-configuration, the last entry in the above example would have overridden the device otherwise assigned to unit #3.

You can use the Filer's Volumes command to determine what units are being used. However, remember that some devices have a second unit number assigned for an alternate DAM that the Volumes command does not display. For example, the device which has the unit #3 (LIF DAM) entry also has the unit #43 (UCSD DAM) entry.

Bubbles as the System Volume

The Pascal 3.0 TABLE program *already* contains code to support BUBBLE memory as a default system volume. This support is declared in the `{system unit auto-search declarations}` constants near the end of the "options" module. This constant tells TABLE to search through 7 possible system volumes. Note that unit number 42 (the default for the Bubble card) is included in the list. If you used a unit number other than 42 for the Bubble card, be sure to change the unit number in the search list above.

Compiling CTABLE

After all modifications have been made to the CTABLE program, the program must be compiled. The resulting code file will be linked and stored as TABLE on the new BOOT: disc.

If you do not know how to compile a Pascal program, see the Compiler chapter for details. The linking procedure is described next.

Linking CTABLE

Once CTABLE.TEXT has been compiled to CTABLE.CODE, the Librarian can be used to create a linked version of CTABLE that will easily fit on the new BOOT: disc.

The following steps assume the program has been compiled and resides in unit #3 as CTABLE-.CODE. Since the linked version of CTABLE is usually less than 15K bytes, it will be put on the same disc that contains the original CTABLE.CODE file (probably CONFIG:) and will later be copied to the new BOOT: disc. If you have two drives, you may wish to put the linked (Output) file directly onto the new BOOT: disc.

1. Press `L` to invoke the Librarian. You may have to temporarily swap discs if the Librarian is not on-line.
2. Press `I` and enter `#3:CTABLE` as the Input file. The Librarian will add the `.CODE` suffix.
3. Press `H` to specify a new Header size; enter a size of 18. (Setting the header size is similar to specifying the directory size of a disc).
4. Press `O` and enter `#3:TABLE.` as the Output file name. The trailing period will suppress the `.CODE` suffix.

5. Perform the actual linking.Syntax:
    a. L – to Link. This will update the display.
    b. D – to toggle the Define source (export text) to NO.
    c. A – to transfer All modules.
    d. L – to finish Linking.
    e. K – to Keep the Output file.
    f. Q – to Quit the Librarian.

6. Copy the linked TABLE to the new BOOT: disc created earlier. Also copy SYSTEM_P and STARTUP to the new BOOT: disc. The new INITLIB that you created earlier should already be on the new BOOT: disc.

   If you did **not** include the BUBBLE module in the INITLIB, the File System will not recognize the Bubble card until you execute the BUBBLE module.

7. Re-boot the system using the new BOOT: disc. Pascal will now recognize the Bubble card.

Bubble Cards in the File System

After the BUBBLE module is installed and an appropriate TABLE program is executed, the Bubble card appears to the File System as a non-removable blocked device (mass storage volume). Any of the local mass storage directory access methods (DAMs) may be used, however the LIF DAM is recommended to allow use of the unit as a boot device.

Executing the Filer's Volumes command will now show that a unit number has been assigned to the Bubble card. For example:

```
Volumes on line:
  1   CONSOLE:
  2   SYSTERM:
  3 # ACCESS:
  4 * SYSVOL:
  6   PRINTER:
 42 # VBUB:
Prefix is - ACCESS:
```

Unlike discs, Bubble memory units are initialized with the LIF DAM before being shipped. This means there is already a directory on the Bubble media. Use the Filer's List command to see the directory. For example, from the Main Command Level, press F to access the Filer, and then use the List directory command by pressing L. Specify #42 (or whatever unit number is assigned to Bubbles). Here is a typical display:

```
VBUB:               Directory type= LIF level 1
created 14-Apr-84 16.21.25 block size=256
Storage order
...file name....     # blks     # bytes   last chng

FILES shown=0 allocated=0 unallocated=8
BLOCKS (256 bytes) used=0 unused=509 largest space=509
```

You can now use the Bubble card as you would any other LIF mass storage volume. It can be zeroed (all files removed) by the Filer's Zero command and it can be initialized by the MEDIAINIT program supplied with the system.

The Bubble Memory cards have access and timing characteristics similar to the HP 9826/36 Computers' internal mini-floppy mass storage drives.

Your Bubble card should provide years of reliable, error-free operation. If you ever have cause to suspect the reliability of the Bubble card, make a back-up copy and and then try re-initializing the card before contacting your Sales and Service Office.

Error Correction

The Bubble Memory unit shipped to you has automatic error correction enabled. If some other memory unit (the hardware package containing the magnetic bubbles) is ever installed in the Bubble card, it should first be initialized by MEDIAINIT to ensure that automatic error correction is enabled.

The Bubble Device

The Bubble Memory unit installed in the Bubble card is a very stable non-volatile storage system. It is not easily damaged by external magnetic fields or mechanical abuse. It is, however, *strongly* recommended that the memory unit not be removed from the card. Removal of the memory unit from the card may damage the "boot loop" or the "seed bubbles".

The boot loop of a Bubble card is equivalent to the spared tracks record of a disc. If the boot loop is damaged, the memory will not function properly. A damaged boot loop may appear as permanent read/write errors, or more likely it will be detected by the TM (Transfer Method) when a UNITCLEAR operation is performed and reported as bad hardware. UNITCLEAR is performed on all units by the TABLE program and by a CLEAR I/O operation initiated from the keyboard (using the (Stop) key).

The memory of a Bubble unit is organized in tracks similar to a disc. Since a bit of information is indicated by the presence or absence of a bubble, information is written to a track by destroying or creating magnetic bubbles.

A magnetic bubble is created by spliting a seed bubble. If the bubble unit is removed or improperly installed a seed bubble may be destroyed or lost. This condition will appear as permanent read/write errors. If you suspect your bubble unit has been damaged in this way, contact your HP Sales and Service Office.

Initialization

The MEDIAINIT program on the ACCESS: disc is capable of initializing a BUBBLE device. The initialization process writes blanks to every location on the media, then writes a default directory to the unit. The only time MEDIAINIT should have to be used is when a Bubble Memory device not supplied by HP is placed on the card.

The Filer's Zero command can be used to remove all the files in the Bubble card. The procedure is similar to the Zero operation of discs. You can change the volume name and the number of directory entries but you should accept the default value for the size of the media.

If you do choose to initialize the Bubble card, execute the MEDIAINIT program and supply the appropriate unit number. Use the default value for all questions.

Interrupts and Overlapped I/O

Bubble devices require immediate interrupt service of relatively short duration. Since the Pascal Workstation File System performs *only* serial I/O, the problem of interrupt priority selection is reduced to ensuring that the BUBBLE module is placed in INITLIB after all other driver modules (but before module LAST). This will ensure that Bubble cards are checked before any other devices (on the same interrupt level) and therefore minimize the time required to service an interrupt.

When performing overlapped Bubble-card-to-Bubble-card transfers, best results are achieved when the destination priority is lower than, or the same as, the source priority. A priority configuration other than this will result in even poorer performance than if non-overlapped I/O is used because the two devices interfere with each other and cause several re-tries per transfer. This is not a problem on machines equipped with cache-memory hardware.

Using EPROM Memory

This section introduces you to the programming and operating characteristics of the HP 98255 EPROM card and the HP 98253 Programmer card. With these cards and Pascal 3.0, you can copy files and volumes into EPROMs.

Overview

EPROMs are high-speed memory devices used for storing programs or other information. The HP 98255 EPROM card and the HP 98253 Programming card support 2764, 27128, and equivalent types of EPROMs.

The EPROM devices are not supplied with the EPROM card. You will have to purchase them separately through an electronic-supply vendor or other source. You probably will also need to purchase an ultra-violet (UV) light source to erase the EPROMs.

The storage capacity of an EPROM can be determined by the final digits of the part number. For example, a 2764-type device contains 64 Kbits (65 536 bits) while a 27128-type device contains 128 Kbits (131 072 bits). Up to 16 EPROMs can be placed on one card; this means that one card provides 131 072 bytes of storage using the 2764-type EPROMs or 262 144 bytes using 27128-type EPROMs.

The data in an EPROM can be read just like RAM memory, however, a special process is needed to program (write) the data into EPROMs. The HP 98253 Programmer card is used for this purpose. An EPROM is programmed by applying a short "burn" pulse while the data being programmed is applied to the output pins. The timing and control of this operation is handled by the Programmer card. Once the EPROMs have been programmed, the Programmer card is no longer needed in the system and can be removed. (With the power turned-off of course!)

An EPROM can be erased (all bits set to "1") by exposing it to a high level of ultra-violet light. Once erased, the EPROMs can be reprogrammed with new data. Check the EPROM manufacturer's specifications for details on the type of UV light source needed and the recommended exposure time.

Configuration Changes Required

There are two changes you need to make to the "standard" configuration in order to use EPROMs:

- Add module(s) to INITLIB.
- Modify the TABLE source program (CTABLE.TEXT)

You may also need to set switches on the cards and install EPROM devices.

INITLIB Driver Modules

In addition to supporting the operations of the HP 98255 EPROM card and the HP 98253 Programmer card, Pascal 3.0 supports the use of EPROMs as a mass storage volume. Transferring a volume into EPROMs would create what could be called an "Eprom-DISC" or "EDISC".

The support modules include:

- The EPROMS module is included on the CONFIG: disc. The module may be installed by either executing it or by using the Librarian to include it in INITLIB.
- The EDRIVER module, also included on the CONFIG: disc, provides *read/write* capability for performing various operations with an EPROM and Programmer card pair. The EDRIVER module can be "P-loaded" or linked to an application program when read/write capability is needed.
- The EPROM Transfer Utility (ETU.CODE) included on the ACCESS: disc allows mass storage volumes to be transferred to EPROMs. When environmental conditions limit the reliability of floppy discs, or when it is desirable to have quick access to commonly used programs or data, a copy of a mass storage volume can be transferred to EPROMs. Transferring a volume to EPROMs creates an "EDISC".

  ETU can also be used to transfer DATA, ASC, and TEXT files to EPROMs. This capability allows arbitrary bit-patterns to be transferred to EPROMs.
- The CTABLE.TEXT file on the CONFIG: disc contains a "template" section to assign unit numbers to EDISCs.
- The Pascal 3.0 IODECLARATIONS module recognizes a Programmer card as CARD_TYPE = 9 (a field of the ISC_TABLE array). This same version also recognizes Bubble memory cards.

You do not have to load any modules before using the ETU program since it already has the necessary drivers included in its code. When you are finished programming the EPROMs, the EPROM module should be added to INITLIB to provide access to EPROMs. This process is described later in this chapter.

Programmer Card Installation

If you have not already installed the Programmer card, see its installation manual for complete details. Some of the installation information is repeated here for convenience.

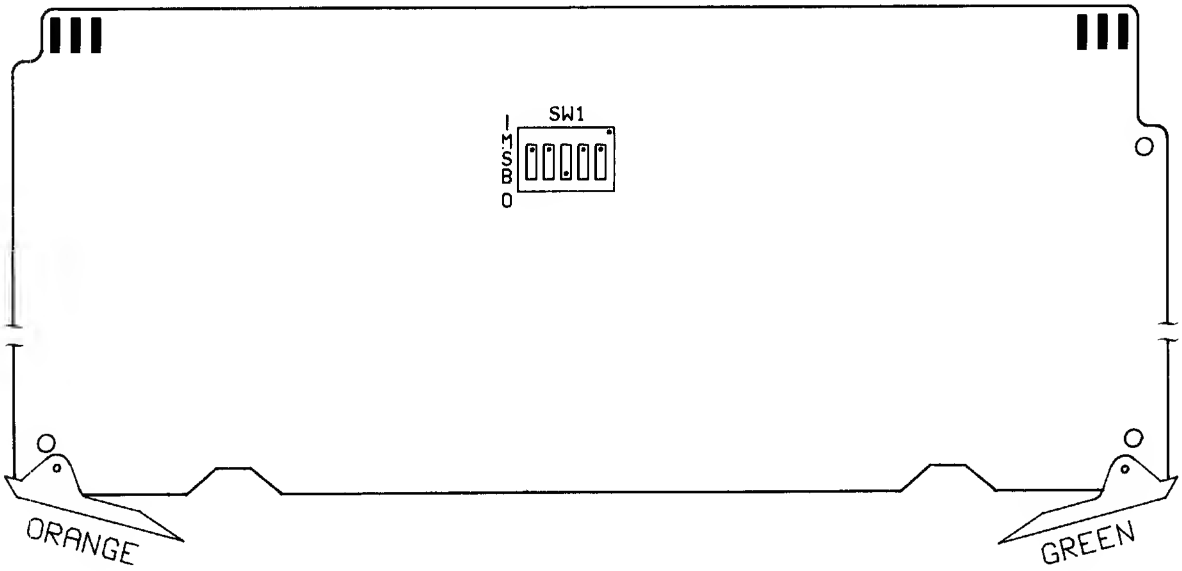
The purpose of the HP 98253 Programmer card is to program (write) information into the EPROMs on the HP 98255 EPROM card. Once the information has been programmed, the Programmer card can be removed from the computer's backplane.

CAUTION

ALWAYS TURN THE COMPUTER OFF BEFORE INSTALLING OR REMOVING INTERFACES.

Perform the following steps to install the Programmer card:

1. Check the select code switch on the Programmer card. The HP 98253 Programmer card's select code has been preset to 27 at the factory. If this conflicts with any other I/O card in the system then change it to an unused select code. If more than one Programmer card is installed, set each card to a unique select code.



Programmer Card Switch Location

Select Code Switch Settings

Switch Settings MSB 43210 LSB	Select Code	Switch Settings MSB 43210 LSB	Select Code
01000	8	10100	20
01001	9	10101	21
01010	10	10110	22
01011	11	10111	23
01100	12	11000	24
01101	13	11001	25
01110	14	11010	26
01111	15	**11011**	**27**
10000	16	11100	28
10001	17	11101	29
10010	18	11110	30
10011	19	11111	31

2. With the computer power turned *off*, install the Programmer card in the computer's backplane. The Programmer card's ribbon cable will be connected to an EPROM card later.

When more than one Programmer card or EPROM card is installed at the same time, the ribbon cable can be connected to different EPROM cards without turning off system power. Be sure that no read or write operation is taking place when the cable is exchanged.

CAUTION

THE PROGRAMMER CARD'S CABLE MUST **NOT** BE REMOVED OR CONNECTED WHEN THE EPROM CARD IS IN USE.

A small light-emitting diode (LED) on the Programmer card indicates when system power is on. (It does **not** indicate when the card is in use.)

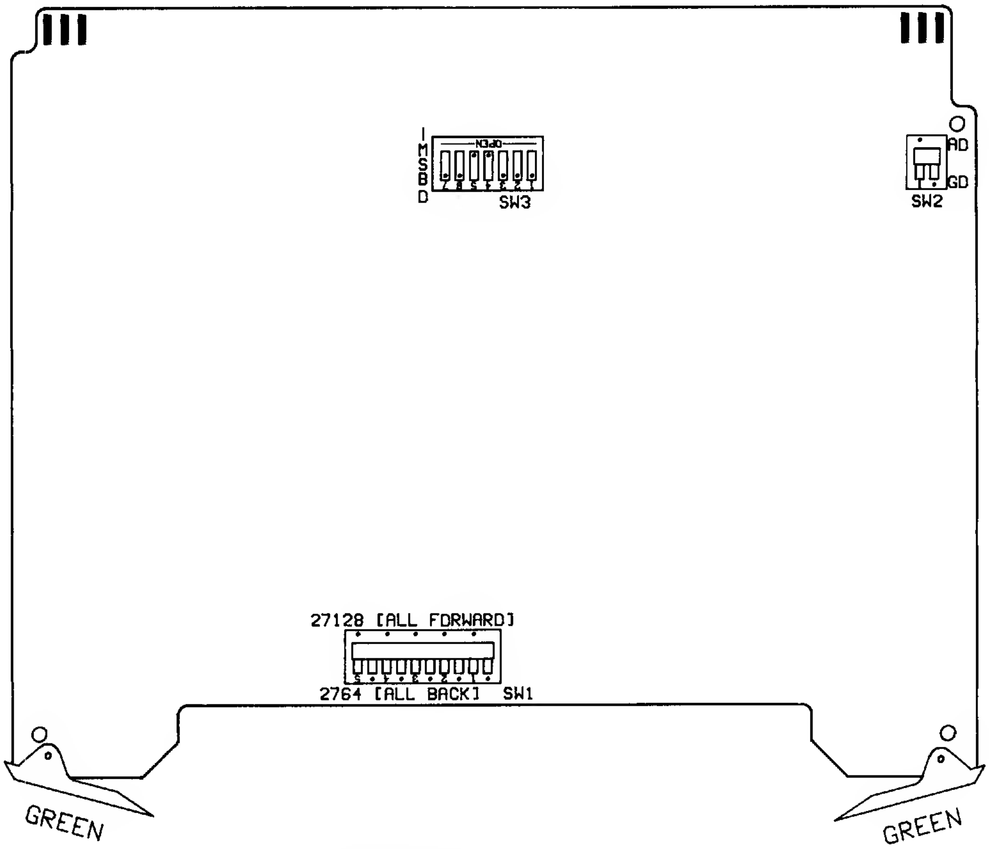
EPROM Card Installation

If you have not already installed the EPROM card, see its installation manual for complete details. Some of the installation information is repeated here for convenience.

Before installing an EPROM card in the computer's backplane, you need to check and set the card switches. There are three sets of switched on the card.

- EPROM-type switch (SW1)
- Address-response switch (SW2)
- Address switch (SW3)

The position of these switches is shown in the following illustration:



EPROM Card Switch Locations

The largest switch is the "EPROM-type" switch. It tells the card's hardware what capacity of EPROM to expect. All segments of this switch are "ganged" together to configure all 16 sockets for either 2764-type or 27128-type EPROMs. You cannot mix two different types of EPROMs on one card, but you do not need to completely fill all 16 sockets with EPROMs. If you only partially fill the card, use pairs of EPROMs (upper and lower byte socket-pairs) and fill the lowest numbered sockets first.

The smallest switch on the card is the "DTACK" switch and it controls the card's response when it is addressed (i.e. whether it should respond like ROM or RAM memory). This switch, which can be set for AD (Automatic DTACK) or GD (Generate DTACK), must be set to AD for the EPROM card to appear in the computer's ROM memory space.

Note

The modules provided with Pascal 3.0 only support EPROM cards which are addressed in the ROM address space. Set the "DTACK" switch to AD (Auto-DTACK).

The third switch determines the base memory address of the card. Special care must be taken to ensure that the address space selected does not overlap another EPROM card or ROM card. The EPROMs on the card are "memory mapped" (in pairs) by ascending socket number. For example, byte 0 is the first location in socket 0U, byte 1 is the first location in socket 0L, byte 2 is the second location in socket 0U, byte 3 is the second location in socket 0L, and so on.

Note

If you have a ROM-based language system, do **not** set the EPROM card's switches to the same address space used by the ROM Language. For instance, the ROM version of the HPL Language System is addressed at $10 0000 and extends up to $12 0000. The ROM 1.0 version of the BASIC Language is addressed at $2 0000 and extends to $2 4000, while the ROM 2.x versions begin at $8 0000 and vary in size.

To see where these ROM-based systems reside, you can check for the presence of "ROM headers" (contents $F0FF) which are located on 16 Kbyte boundaries, beginning at $2 0000 and extending through the Auto-DTACK range of addresses. Auto-DTACK extends to $20 0000 for cache-memory processor boards (i.e., machines with "U" suffix such as the HP 9836U) and $40 0000 for non-cache-memory processor boards (such as the HP 9836A).

Although the switches can be set to make the EPROM card appear almost anywhere in the computer's address space, the following table shows the recommended settings. When the smaller capacity EPROMs are used, multiple cards can be addressed 128 Kbytes apart; cards filled with the larger capacity devices must be addressed 256 Kbytes apart.

Address Switch Settings

Switch Settings	EPROM Card's Base Address for Programming		
MSB LSB	Hex Start Address	Decimal Address (2764-type devices)	Decimal Address (27128-type devices)
0 0 0 0 0 0 1	$2 0000	131 072	
0 0 0 0 0 1 0	$4 0000	262 144	262 144
0 0 0 0 0 1 1	$6 0000	393 216	
0 0 0 0 1 0 0	$8 0000	524 288	524 288
0 0 0 0 1 0 1	$A 0000	655 360	
0 0 0 0 1 1 0	$C 0000	786 432	786 432
0 0 0 0 1 1 1	$E 0000	917 504	
0 0 0 1 0 0 0	$10 0000	1 048 576	1 048 576
0 0 0 1 0 0 1	$12 0000	1 179 648	
0 0 0 1 0 1 0	$14 0000	1 310 720	1 310 720
0 0 0 1 0 1 1	$16 0000	1 441 792	
0 0 0 1 1 0 0	$18 0000	1 572 864	1 572 864
0 0 0 1 1 0 1	$1A 0000	1 703 936	
0 0 0 1 1 1 0	$1C 0000	1 835 008	1 835 008
0 0 0 1 1 1 1	$1E 0000	1 966 080	
0 0 1 0 0 0 0	$20 0000	2 097 152	2 097 152
0 0 1 0 0 0 1	$22 0000	2 228 224	
0 0 1 0 0 1 0	$24 0000	2 359 296	2 359 296
0 0 1 0 0 1 1	$26 0000	2 490 368	
0 0 1 0 1 0 0	$28 0000	2 621 440	2 621 440
0 0 1 0 1 0 1	$2A 0000	2 752 512	
0 0 1 0 1 1 0	$2C 0000	2 883 584	2 883 584
0 0 1 0 1 1 1	$2E 0000	3 014 656	
0 0 1 1 0 0 0	$30 0000	3 145 728	3 145 728
0 0 1 1 0 0 1	$32 0000	3 276 800	
0 0 1 1 0 1 0	$34 0000	3 407 872	3 407 872
0 0 1 1 0 1 1	$36 0000	3 538 944	
0 0 1 1 1 0 0	$38 0000	3 670 016	3 670 016
0 0 1 1 1 0 1	$3A 0000	3 801 088	
0 0 1 1 1 1 0	$3C 0000	3 932 160	3 932 160
0 0 1 1 1 1 1	$3E 0000	4 063 232	

Once the EPROM card's switches have been set, install the EPROM devices on the HP 98255 EPROM card. Be very careful when installing the EPROMs on the card, since the pins are easily bent. Both the EPROMs and the sockets have notches to indicate the proper orientation. See the installation manual for details.

With the power switched *off*, install the EPROM card in the computer's backplane.

Multiple EPROM Cards

If more than one blank EPROM card is installed in the computer's backplane at the same time, be sure each EPROM card is addressed to different memory locations. The lowest addressed card should be programmed first. Blank EPROM cards can not be detected by the Pascal system unless they are connected to the Programmer card.

Cable Connections

When you have finished installing the Programmer and EPROM cards, you can connect the ribbon cable from the Programmer card to the desired EPROM card. The cable connection defines and establishes the "card-pair" for programming operations.

The Programming Utility

The ETU program supplied with Pascal 3.0 supports the following operations for an EPROM and Programmer card-pair:

- Display current Programmer and EPROM card information
- Check for blank space on the EPROM card
- Transfer a mass storage volume to EPROM (EDISC creation)
- Transfer DATA, TEXT, or ASC files to EPROM (user-defined patterns)

The exact action taken in a transfer operation depends on the type of file involved. All transfers are done in two passes through the data. The two passes perform the same actions except that the data is actually programmed (written) only during the second pass.

Note

All file types other than TEXT and ASC are treated as DATA files.

Transferring Volumes to EPROM

When you specify that a volume is to be transferred to EPROM, the ETU program assumes that an EDISC is to be created. The EDISC will appear to the File System as a mass storage volume not unlike a floppy disc, but with much faster access. The maximum size of the volume depends on the capacity of the EPROMs installed in the card. The largest EDISC that can be created contains 256K bytes, since EDISCs can not cross EPROM card boundaries.

Once an EDISC has been created, you should not copy the EDISC to any other mass storage volume.

Booting from EDISC

Boot ROM 3.0 and later versions can boot from an EDISC. Booting from these devices is like booting from any other mass storage media; the system is copied into RAM and executed from there rather than from the EDISC (unlike ROM-based systems which execute from ROM).

EDISC as the System Volume

Pascal can also recognize an EDISC volume as the system volume; however, since the system volume is used by the system to store all temporary files, I/O error 18 ("Device is write-protected") will be reported whenever the system attempts to write to this "write-protected" device.

AUTOSTART and other normal stream files will not work if the system volume is an EDISC. (Normally, when a file is Streamed, the file is copied to the file named STREAM on the current system volume; this is not possible with EDISCS, since they are effectively "write-protected.") You should use the AUTOKEYS file to perform autostart functions from these devices. Other stream file names must contain a [*] specifier which indicates that the stream-file prompt feature is disabled. See the description of the Stream command in the Overview chapter for further details of using prompts in Stream files.

EDISC Headers

When a volume is transferred to EPROMs, an EDISC "header" is first generated and programmed into the EPROMs. The Boot ROM can detect the header and make that information available to the file system. In other words, if a volume is transferred to EPROMs, special information is added that allows the Boot ROM to detect and possibly boot from the EDISC.

Boot ROM 3.0 and later versions check for EDISC headers (and other types of headers) on 16 Kbyte boundaries, starting at 128 Kbytes ($2 0000) and continuing through the Auto-DTACK range of addresses; this range extends to $20 0000 for machines with cache-memory processor boards (i.e., computers with "U" suffix such as the HP 9836U), and $40 0000 for machines without cache-memory processor boards (such as the HP 9836A). This searching operation effectively divides the address space into 16 Kbyte "blocks".

Since the Boot ROM will check for an EDISC header on every 16 Kbyte block boundary, more than one EDISC can be programmed onto a single EPROM card. There are 8 blocks (numbered 0..7) on an EPROM card using the small capacity EPROMs and 16 blocks (numbered 0..15) using the large capacity EPROMs.

To prevent the Boot ROM from accidentally interpreting the contents of a block boundary as an EDISC header, the utility program writes binary zeros (hexadecimal pattern $0000) into the boundary locations searched by the Boot ROM. The volume's data is appropriately mapped around the block boundaries. The mapping operation is completely handled by the system, but this does mean the EDISC volume will be a few bytes larger than the original volume.

The total number of bytes needed to program a volume can be computed by taking the source volume's size and adding 18 bytes for the EDISC header and 2 bytes for each 16 Kbyte boundary crossed. If the last sector of the volume is unused, the extra bytes can be truncated without loss of data.

Transferring Files to EPROM

The ETU program can be used to transfer DATA, TEXT, or ASC type files to EPROMs. If the file type is **not** TEXT or ASC, the files will be programmed into EPROMs so as to create an exact bit for bit copy. If the file type is TEXT or ASC then *only the data parts* are programmed into EPROM (not the data separators and other "overhead"; this is equivalent to reading a line from the file into a string with a READLN statement and then "burning" the contents of the string.)

Unlike volumes, individual files transferred to EPROMs without the "directory" information of a volume cannot be detected by the file system. If you write a program to access a file that was programmed into EPROMs, you will have to tell your program where to find it. Even if only one file is to be transferred to EPROMs, you might consider putting the file in a volume and transferring the volume.

Not only do you have to keep track of the location of individual files transferred to EPROM, you must be sure that the data does not accidentally appear to the Boot ROM as a "ROM header". The Boot ROM searches for a two-byte header pattern (F0FF hexadecimal) at 16K byte intervals in the Series 200 Computer's ROM space.

The header pattern is not likely to occur in TEXT or ASCII files, however, a DATA file programmed into EPROMs may contain such a bit-pattern, and if the pattern occurs on a 16 Kbyte "block" boundary, unpredictable results may occur. The ETU program does not check for this condition.

Preparing a Transfer

Before starting the utility to transfer a volume or file to EPROMs, you must decide what you want to transfer. There are some restrictions that may influence your decision.

- The total number of bytes transferred must be less than the total capacity of the EPROMs. Excess bytes will be truncated. It is unlikely that a truncated file will be very useful.
- If the "source" volume is larger than the current available space on the EPROM card, the volume will be truncated. Since LIF volumes contain all of their directory information at the beginning of the volume, you can truncate the unused sectors at the end of a volume with relative impunity.

For the purposes of this discussion, it has been decided to transfer the Pascal Editor and Filer to EPROMs. This will allow fast access to the programs without requiring as much RAM memory as is necessary to "P-load" both of them. The number of bytes required for both the Editor and Filer is less than 120K bytes so both programs will easily fit on the EPROM card even if the smaller capacity EPROMs are installed.

Note that programs are **not** usually executed in EPROMs; rather, a copy of the program is made in RAM memory and then the copy is executed. When you quit the program, and the copy is no longer needed, the RAM memory used for the copy is free to be used by other programs. This has advantages over a program that is "P-loaded" since a "P-loaded" program remains in RAM memory until the next boot operation.

The volume containing the programs to be transferred to EPROMs should not be any larger than necessary since the entire volume will be transferred, including any unused sectors. Once the volume has been transferred to EPROMs, there is no way to go back and fill the unused sectors in the volume. Therefore, for our example, the best approach will be to create a memory volume just large enough to hold both the Editor and Filer. This will be the volume that is transferred to EPROMs.

Creating a Memory Volume

A memory volume needs two (2) "system" sectors, one (1) sector for directory information, and enough sectors to hold the files. The size of the Filer is about 228 sectors (58368 bytes), and the size of the Editor is about 232 sectors (59392 bytes). Thus, in our example, we need 3 + 1 + 460 sectors, or a total of 464 sectors.

The Memvol command "thinks" in 512-byte blocks not in 256-byte sectors. Therefore, to create the correct size memory volume, we need an even number of sectors (round-up). The total number of blocks is then 460/2 or 230 blocks.

From Pascal's Main Command Level, press M to create a memory volume. Answer 230 to the "Number of Blocks" question, and answer 8 to the "Number of Directory Entries" question.

When you have created the memory volume, Filecopy the Editor and Filer from the ACCESS: disc. Then use the Filer to Change the volume name from RAM: to ESYS: (the volume name will also be transferred to EPROMs).

The "Empty sockets" command of the transfer utility can be used to protect EPROMs from being programmed if there are a large number of unused sectors in the volume being transferred to EPROMs. The ETU program will then detect that the volume being transferred is larger than the available space and allow you to truncate the unused bytes. Be sure that it does not truncate part of a file!

The EPROM Transfer Utility

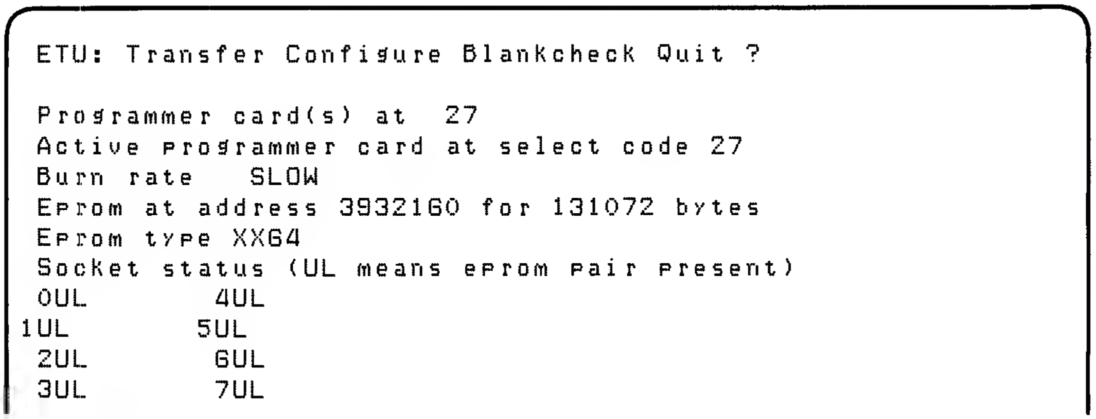
The EPROM Transfer Utility program (ETU.CODE) is included on the ACCESS: disc. This utility provides a convenient method of programming the EPROMs on a HP 98255 card. Either single files or entire volumes can be transferred to EPROMs with this utility.

If you haven't already executed ETU.CODE, do so now. When the utility is executed, it automatically searches for the Programmer card connected to an EPROM card. If a card is missing or incorrectly installed, you will get one of the following messages.

```
*** NO PROGRAMMER CARD IN SYSTEM ***

NO EPROM CARD ATTACHED TO PROGRAMMER CARD
```

If the system does not recognize the Programmer card, turn power off and check the select code switch settings. You should check that each switch segment is toggled correctly and that no other interface card is set to the same select code.

When the Programmer and EPROM cards have been correctly installed and connected to each other by the Programmer card's cable, the main menu is displayed. (Note: the space-bar was pressed to remove the release date and copyright notice from the following display.)



```
 ETU: Transfer Configure Blankcheck Quit ?

 Programmer card(s) at  27
 Active programmer card at select code 27
 Burn rate   SLOW
 Eprom at address 3932160 for 131072 bytes
 Eprom type XX64
 Socket status (UL means eprom pair present)
 0UL        4UL
1UL        5UL
 2UL        6UL
 3UL        7UL
```

There are four functions available from the main menu: Transfer, Configure, Blank check, and Quit. Each of these functions will be explained on the following pages.

Your display may differ depending on the select code setting of the Programmer card and the capacity of EPROMs installed in the EPROM card. If more than one Programmer card is installed in the system, all operations will use the "active" Programmer card. If more than one EPROM card is installed in the system, all ETU operations will affect only the EPROM card connected to the (active) Programmer card's cable.

The various functions are activated by typing the first letter of the appropriate operation (for example, C for Configure). Lettercase does not matter. Incorrect letters are ignored except if the program is under stream control. When streaming, incorrect letters will abort the program and the stream file.

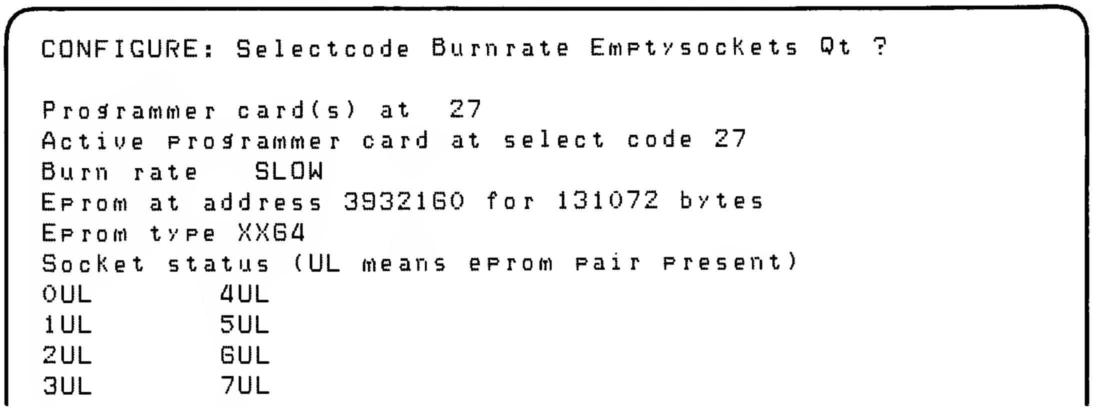
All operations can be aborted by typing Shift-Select (SHIFT-EXECUTE) for single character answers or Shift-Select and then Return or Enter (SHIFT-EXECUTE and then ENTER) for multi-character answers.

In stream file operations, answers to optional questions are automatic and are the *first* option given in the prompt. For example:

- For a YES/NO question ending with "(Y/N) ?" – The stream answer is "Y"
- For an ABORT/TRUNCATE question ending with "(A/T) ?" – The stream answer is "A"

Configuration

From the main menu, press C to display the configuration sub-menu. The main menu is replaced with a sub-menu which lets you change the select code, the burn rate, and specify any empty EPROM sockets.



```
CONFIGURE: Selectcode Burnrate Emptysockets Qt ?

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL        4UL
1UL        5UL
2UL        6UL
3UL        7UL
```

The configuration functions are explained next. Pressing Q will return you to the main menu.

Select Code

When only one Programmer card is in the system, it is automatically chosen as the active Programmer card and the select code is properly set.

If you have more than one Programmer card installed in the system and wish to change operations to a different Programmer card, press S for Select code. The following prompt will appear at the bottom of the display:

```
New select code (27) ?
```

The number in parentheses indicates the select code of the currently selected Programmer card. You may either press Return or ENTER to accept the current select code or type the select code of a different Programmer card. If the select code you type is valid, the display will be updated with the new information. An error message will be displayed if the new select code does not correspond to a Programmer card.

Burn Rate

Pressing B will cause the Burn rate to change from SLOW to FAST or from FAST to SLOW (the display is automatically updated).

> **Note**
>
> All EPROMs may be programmed at the slow burn rate. Some EPROMs are not guaranteed to retain the pattern if the faster rate is used. Check the EPROM manufacturer's specifications before using the faster programming rate.

If the FAST burn rate is specified and a location fails to accept the data, the burn rate will automatically be switched to SLOW.

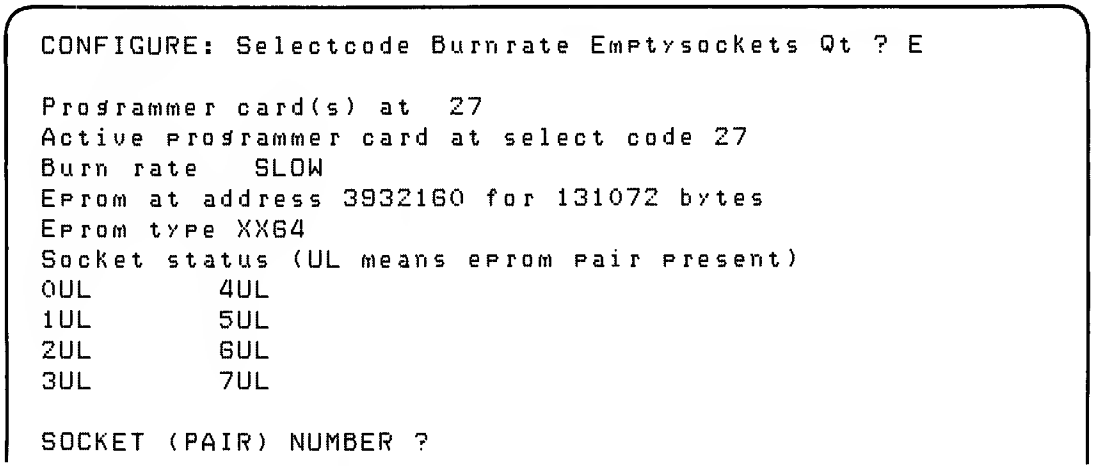
The FAST burn rate programs at 13.1 ms/byte while the SLOW burn rate programs at 52.3 ms/byte. The card circuitry can program both upper (even address) and lower (odd address) bytes in parallel so the effective rate is 13.1 or 52.3 ms/word. Therefore, programming every location in a full set of large capacity EPROMs using the FAST burn rate will take about an hour.

Note that the Burn rate is a global attribute not associated with a particular Programmer or EPROM card.

Empty Sockets

An empty EPROM socket appears to be an erased (blank) EPROM. This condition can not be detected until an attempt is made to program a pattern into such a location.

Pressing E allows you to specify which sockets of an EPROM card do not contain EPROMS; the information is used in the calculation of the capacity of an EPROM card. EPROMs must be used in pairs and up to 8 pairs of EPROMs may be used in one card.



```
CONFIGURE: Selectcode Burnrate Emptysockets Qt ? E

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL       4UL
1UL       5UL
2UL       6UL
3UL       7UL

SOCKET (PAIR) NUMBER ?
```

For example, if you answered "7" to this question, the display would be updated as follows:

```
Socket status (UL means eprom pair present)
0UL        4UL
1UL        5UL
2UL        6UL
3UL        7 empty
```

In this manner, you can specify all of the empty sockets. If you make a mistake, re-execute the command with the same socket number; the program will again mark the socket pair as usable.

An error message is displayed if the socket pair number is out of range.

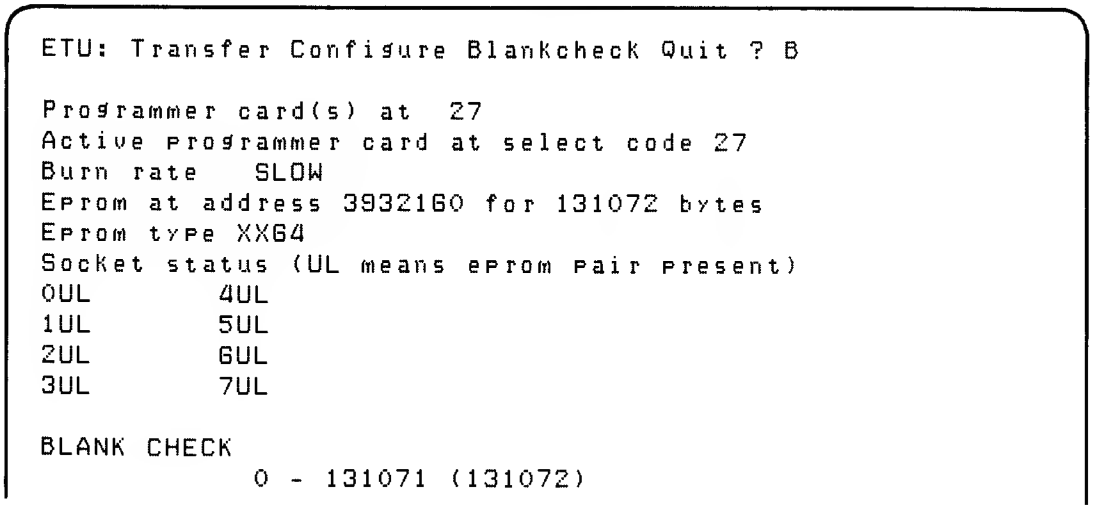
Quitting the Sub-Menu

Quitting the Configure sub-menu will return you to the main menu. Once you have completed the configuration for the active card-pair, the next step is to check for available space in the EPROMs.

Blank Check

Pressing B from the main menu will show the used and unused space (according to how many EPROMs you've told the program are on the EPROM card connected to the active Programmer card). A blank EPROM has all of its bits set to binary 1's. Thus, a blank byte would contain the hexadecimal pattern FF.

Unused space will be shown at the bottom of the display. For example:



```
ETU: Transfer Configure Blankcheck Quit ? B

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL        4UL
1UL        5UL
2UL        6UL
3UL        7UL

BLANK CHECK
            0 - 131071 (131072)
```

The number in parenthesis indicates the size of the unused space (in bytes). The two numbers separated by a hyphen indicate the relative address of the unused space within the active card.

The above display indicates that the entire EPROM card is unused. Bytes 0 through 131 071 are blank and the total number of contiguous blank bytes is 131 072.

If no blank bytes can be found on the current EPROM card, the following error message will be displayed:

```
NO BLANK SPACE FOUND
```

Typically, after an EPROM has been programmed, there will be some bytes containing the hexadecimal pattern: FF. These bytes will appear to the program as "blank" and the Blank Check option will list them as follows:

```
    address - address (size in bytes)
    address - address (size in bytes)
```

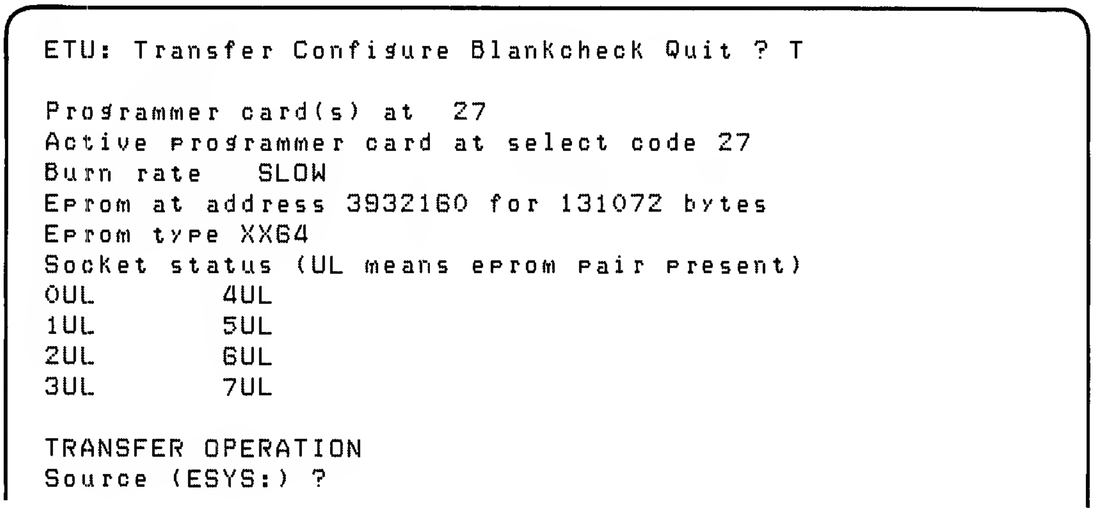
The above lines are repeated as many times as required, in groups of 6, with a prompt to press the spacebar to continue between each group. The addresses given are relative to the base address of the EPROM card. The size is likely to be only a few bytes for addresses that actually contain data. (A hexadecimal FF programmed into EPROM looks like a "blank" location.) The last entry is likely to indicate any truly "blank" space. The sockets you've specified as empty are not counted.

Now that the available space has been determined, you are ready to transfer a file or a volume to EPROMs.

Transfer

Pressing T from the main menu will prompt you for information about the transfer operation. ETU makes some assumptions to try to help you.

The display will show the following:



```
ETU: Transfer Configure Blankcheck Quit ? T

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL       4UL
1UL       5UL
2UL       6UL
3UL       7UL

TRANSFER OPERATION
Source (ESYS:) ?
```

The ETU program assumes that a volume will be transferred to EPROMs. The volume name in parenthesis is the current prefixed volume. You may accept the volume name by pressing Return or you may type another volume name. If you specify both a volume name and a file name then ETU assumes that a single file is to be transferred to EPROMs. If you do specify a different volume or a file, the display will be updated accordingly.

When transferring a volume to an EPROM card, if no "blank" block is found on the EPROM card the following message is given:

```
   *** NO BLANK BLOCK ON THIS EPROM CARD ***
```

The program will then display:

```
Start at eprom block offset (0) ?
```

The value in parenthesis indicates the lowest numbered "blank" block. (If every block has been programmed, a zero is displayed.)

Or if a file was specified:

```
Start at eprom byte offset (0) ?
```

For a file, the value in parenthesis is always zero.

If the default value in parenthesis is acceptable, press Return to begin the transfer operation. Optionally, you may specify a different block offset or byte offset. See the previous sections on Transferring Volumes and Files for the details about offsets.

If there is insufficient space on the EPROM card for the transfer, the ETU program will prompt:

```
DATA EXCEEDS EPROM SPACE BY xxxx BYTES
Abort transfer or Truncate file (A/T) ?
```

Where xxxx represents the number of excess bytes.

A reply of A or Shift-Select (SHIFT-EXECUTE) will cancel the operation. A reply of T will cause the transfer of only as much data as will fit on the EPROM card. If this happens during the execution of a stream file, the transfer operation will abort and the stream file will be terminated.

A transfer is a two-pass operation. The first pass checks the data and the EPROMs. The second pass actually programs the data into the EPROMs and verifies that it has been stored correctly.

Unless an error occurs, the transfer is automatic from here on.

Check Failure

Check failure is detected during the first pass. The byte to be programmed is matched against the byte on the EPROM card. If the EPROM can not be made to contain the new pattern then a CHECK FAIL results. (An EPROM's "0" bits can not be changed to "1" bits.)

```
CHECK FAIL AT ABSOLUTE ADDRESS aaa
BYTE POSITION bbb FROM START LOCATION
EPROM SOCKET un BYTE nn
```

Where "aaa" is the absolute machine address of the byte which will not program or did not program. The position "bbb" is the byte index (from 0) of the byte in the file. The position is also identified by EPROM ("un" is socket identifier: for example, U1 or L4) and "nn" is the byte offset (from 0) within the identified EPROM.

Burn Failure

If an EPROM fails to accept a byte of data using the FAST burn rate, the utility automatically switches to the SLOW burn rate, updates the display, and attempts to continue.

If the burn rate is already SLOW when a byte fails to program properly, then a "BURN FAIL" occurs. The utility is aborted and a message is displayed. For example:

```
BURN FAIL AT ABSOLUTE ADDRESS 3997696
BYTE POSITION 65536 FROM START LOCATION
EPROM SOCKET 4U BYTE 0
```

If the programming fails exactly on a socket boundary ("BYTE 0" in the example above) check to see if the socket is empty or if the EPROM is improperly installed (bent pins).

Quitting ETU

Pressing (Q) from the main menu will quit the utility and exit to the Pascal Main Command Level.

This concludes the operations of the ETU program. Once the EPROMs in an EPROM card have been programmed, the Programmer card can be removed from the system. (With the power switched off of course!)

Before the File System can recognize an EDISC, a Transfer Method (TM) module must be loaded into the system and a modified version of the CTABLE program must be compiled and executed. (The ETU program has its own driver module and could locate the EPROM card since it was connected to the Programmer card.)

Loading the EPROMS Module

The EPROMS module is supplied on the Pascal 3.0 CONFIG: disc. As with other driver modules, there are two ways to load the module:

- Execute it (with the eXecute command at the Main Level)
- Add it to INITLIB and re-boot

Executing the module "permanently" loads the module, but must be performed every time the system is booted. Adding the module to INITLIB eliminates having to load the module each time you re-boot the system.

Adding the EPROMS Module to INITLIB

If you have two disc drives, the creation of the new INITLIB is relatively simple. If you only have one disc drive, you will need to create a memory volume large enough to hold the new library (about 175K bytes).

To create a memory volume, press (M) from the Main Command Level. You will be prompted for the number of 512 byte blocks (answer 350) and the number of directory entries (answer 8). If you are not familiar with memory volumes, see the Memory volume command in the Main Command Reference section of the Overview chapter.

1. Initialize a disc, and then use the Filer's Filecopy command to copy the BOOT: disc onto this disc. Since you will be storing the new INITLIB on this new BOOT: disc, you can Remove the existing INITLIB file from the disc. Since the old INITLIB was probably not the last file on the disc (and the new INITLIB will probably be bigger than the old), you should Krunch the disc.

   (It is a very good practice to create a new BOOT: disc rather than modifying your present BOOT: disc. That way you can always return to where you are, no matter what happens to the new disc.)

2. Invoke the Librarian. This is done by pressing (L) from the Main Command Level. If the Librarian is not on-line, insert the ACCESS: disc and try again. Remove the ACCESS: disc once the Librarian has loaded.
3. Insert the *old* BOOT: disc into Unit #3 and the *new* BOOT: disc into Unit #4. (If you are using a memory volume, the memory volume will be the "blank disc". Use whatever unit number you assigned to the memory volume instead of Unit #4 for the remaining steps.)
4. Now use the Librarian to create the new INITLIB.Syntax:

   a. Press (I) and type `#3:INITLIB.` and (Return) or (ENTER) to enter the Input file. You must include a trailing period to prevent the Librarian from appending the `.CODE` suffix.

      When the Librarian finds the input file, the display will show the name of the first module in the file. You should see the module named KERNEL. If you have a printer, you can press (F) to list all of the modules in INITLIB.

      The EPROMS module can be inserted anywhere after the IODECLARATIONS module but before the module named LAST (it must also precede module BUBBLE, if that module is present). In this example, the module will be included as the next-to-last module in the new INITLIB.

   b. Press (O) and enter `#4:EPLIB.` as the Output file. Again, a trailing period prevents the `.CODE` suffix from being appended to the file name.

      (This disc must *not* be removed until you have finished creating the new EPLIB file.) If you are using a memory volume, use the unit number of the memory volume.

c. Press E to enter the Edit mode. You should now see this prompt (in the middle of the screen):

```
F  First module: KERNEL
U  Until module: (end of file)
```

d. Press U enter `LAST` as the Until module. You can now transfer all modules in the file up to (but not including) module LAST by pressing C.

e. When the preceding transfer is complete, press A to append a module to the EPLIB Output file. The Librarian prompts with `Input file:`. Put the CONFIG: disc, or whichever disc now contains the EPROMS module, in Unit #3 (*not* #4, which must not be removed). Enter `#3:EPROMS.` as the Input file specification.

f. The Librarian now prompts with `Enter list of modules or = for all`. Enter = to specify all modules. After the EPROMS module has been transferred to the EPLIB library, the Librarian prompts with `Append done, <space> to continue`. If you removed the BOOT: disc (or the one that contains the INITLIB Input file) to put in the CONFIG: disc, replace the BOOT: disc now *before* pressing the spacebar to answer the prompt.

(If you removed the BOOT: disc in #3: and did not replace it before pressing the spacebar, you get the following message: `cannot open '#3:INITLIB', ioresult = 10`. In such case, don't worry. Remove the CONFIG: disc and insert the BOOT: disc, then press I and enter `#3:INITLIB.` as the Input file. Press E to return to Edit mode, and go back to where you were previously by pressing F and entering `LAST` as the First module. Proceed with step g below.)

g. Press T to transfer module LAST to the EPLIB file (if you got the error described in the preceding step, press C instead of T). Then press S to stop editing and K to keep the file.

h. You should now verify that the EPROMS module was indeed copied to the Output file. Press I and enter `#4:EPLIB.` as the Input file. Press the spacebar repeatedly to scan through the modules in the new library file. If you have a printer, press F to get a File Directory listing.

i. If all modules are present, then press Q to Quit the Librarian.

5. If you have been using two discs, use the Filer to Change the file named EPLIB (on the new BOOT: disc) to INITLIB. If you used a memory volume, remove the old BOOT: disc from Unit #3 and insert the new BOOT: disc; then use the Filer to Filecopy EPLIB from the memory volume to the new disc, changing the file name to INITLIB in the process.
6. Re-boot the computer, which installs the new INITLIB containing the EPROMS module.

To make the EPROM card(s) available to the File System as mass storage units, the CTABLE program must be modified to reserve an entry in the Unit Table.

CTABLE Modifications

The Pascal 3.0 CTABLE program, supplied on the CONFIG: disc, contains a "template" for EPROM cards. You can either use the Editor's Find command to search through all occurrences of the `EPROM` token until you find the template, or Jump to the end of the program and scroll up until you see the EPROM template shown below.

```
$if false$ { EPROM DISC }
  {watch for conflicting uses of unit 42}
  tea_EPROM(42,primary_dam,{ sequence number } 0);
$end$
```

To activate the template, change `$if false$` to `$if true$` as shown in the following example:

```
$if true $ { EPROM DISC }
  {watch for conflicting uses of unit 42}
  tea_EPROM(42,primary_dam,{ sequence number } 0);
$end$
```

The template assigns the lowest addressed EDISC to Unit 42. It should be noted that this unit number is also the default for Bubble cards and may have to be changed to some other unit number more appropriate to your peripheral configuration.

EDISCs are recognized according to their relative addresses in the ROM address space of the system. The EDISC with the lowest address is assigned sequence number 0, the second lowest is assigned sequence number 1, and so on.

If you have more than one EDISC, your template might appear as follows:

```
$if true $ { EPROM DISC }
  {watch for conflicting uses of unit 42}
  tea_EPROM(42,primary_dam,{ sequence number } 0);
  tea_EPROM(27,primary_dam,{ sequence number } 1);
  tea_EPROM(28,primary_dam,{ sequence number } 2);
  tea_EPROM(31,primary_dam,{ sequence number } 3);
$end$
```

To force recognition of an EDISC (or multiple EDISCs), call the procedure TEA_EPROM with the appropriate unit number, DAM identifier, and sequence number.

The connection between unit number and address is made when a CLEARUNIT call is made to the TM. This implies that if the address switches of the EPROM cards are changed, the cards may be assigned different Unit Table entries.

In the Unit Table, the SC field is -1 and the sequence number is stored in the DV field.

Compiling CTABLE

Once the necessary modifications have been made to the CTABLE program, the program should be compiled and the resulting CODE file linked (export text removed) and stored as TABLE on a BOOT: disc. If you do not know how to compile a program, see the Compiler chapter. The linking operation is described next.

Linking CTABLE

Once CTABLE.TEXT has been compiled to CTABLE.CODE, the Librarian can be used to create a linked version of CTABLE that will easily fit on the new BOOT: disc.

The following steps assume the program has been compiled as CTABLE.CODE on unit #3. Since the linked version of CTABLE is usually less than 15K bytes, it will be put on the same disc that contains the CTABLE.CODE file and will later be copied to the new BOOT: disc. If you have two drives, you may wish to put the linked (output) file directly onto the new BOOT: disc.

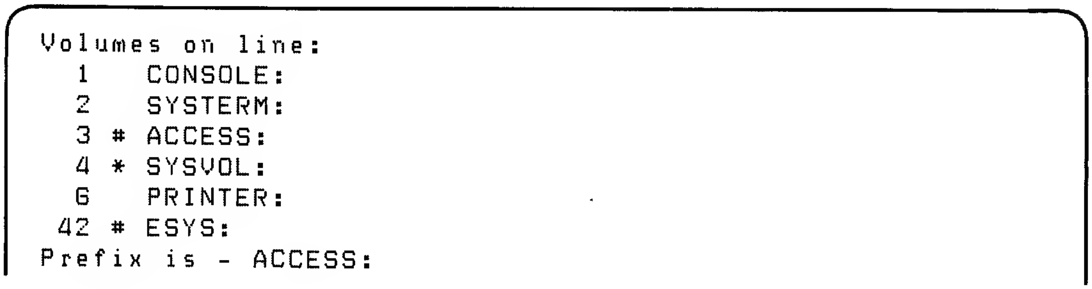
1. Press `L` to invoke the Librarian. You may have to temporarily swap discs if the Librarian is not on-line.
2. Press `I` and enter `#3:CTABLE` as the Input file. The Librarian will add the `.CODE` suffix.
3. Press `H` to specify a new header size. Enter a size of 18. (Setting the header size is similar to specifying the directory size of a disc).
4. Press `O` and enter `#3:TABLE.` as the Output file. The trailing period will suppress the `.CODE` suffix.
5. Perform the actual linking.Syntax:
   a. `L` – to Link. This will update the display.
   b. `D` – to toggle the define source (export text) to NO.
   c. `A` – to transfer all modules.
   d. `L` – to finish linking.
   e. `K` – to keep the output file.
   f. `Q` – to quit the Librarian.
6. Copy the linked TABLE to the new BOOT: disc created earlier. Also copy SYSTEM_P and STARTUP to the new BOOT: disc. The new INITLIB that you created earlier should already be on the new BOOT: disc.

   If you did *not* include the EPROMS module in the INITLIB, the Pascal file system will not recognize the EPROM card until you install the EPROMS module.
7. Re-boot the system using the new BOOT: disc. The File System will now recognize the EPROM card.

EPROM Cards in the File System

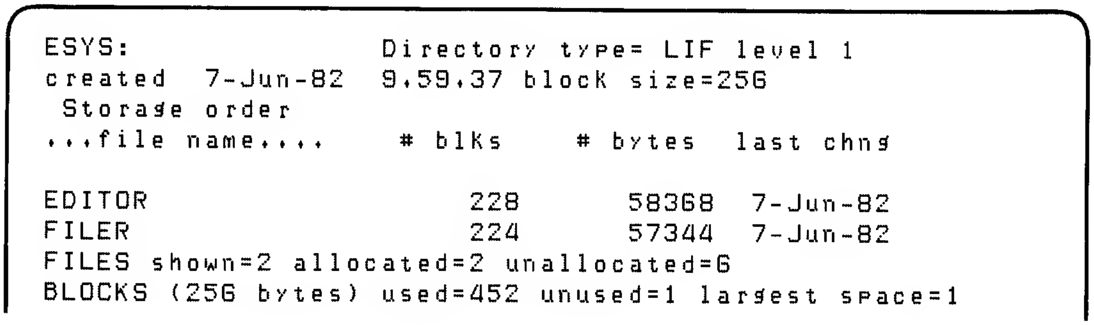
After the necessary modifications have been made, and the system re-booted, you can use the Filer's Volumes command to see an EDISC.

For example:



```
Volumes on line:
  1   CONSOLE:
  2   SYSTERM:
  3 # ACCESS:
  4 * SYSVOL:
  6   PRINTER:
 42 # ESYS:
Prefix is - ACCESS:
```

Use the Filer's List command to see the directory. For example:



```
ESYS:                    Directory type= LIF level 1
created  7-Jun-82        9.59.37 block size=256
 Storage order
...file name....          # blks     # bytes   last chng

EDITOR                         228       58368  7-Jun-82
FILER                          224       57344  7-Jun-82
FILES shown=2 allocated=2 unallocated=6
BLOCKS (256 bytes) used=452 unused=1 largest space=1
```

You may now use EPROMs as you would any other write-protected mass storage volume. Remember, an EDISC should not be copied to another mass storage volume.

This concludes the EPROM installation and programming information. The remainder of this chapter covers the support modules for EPROMs.

Using DC600 Tapes

This section describes use of the DC600 Streaming Tape Drives, such as the HP 9144, for mass storage operations. If you have one of the Command Set '80 Series Disc Drives, you may also have a DC600 tape cartridge drive integrated into the machine for backup.

Tape Drives Supported

The currently supported DC600 Tape and CS80 Disc/Tape Drives include the following HP products:

- HP 9144
- HP 7908
- HP 7911
- HP 7912
- HP 7914

Tape Lengths

There are two lengths of DC600 tapes: 150 feet and 600 feet; these tapes have capacities of 17 and 67 Mbytes, respectively. Both tapes can be directly accessed by the Pascal File System.

Tape Access Methods

The Pascal system provides two methods of computer-controlled tape access. The first is a utility program with capabilities similar to the integrated disc/tape product's "switch" backup. The Operating and Installation Manual that came with the product describes a method of off-line "switch" backup, involving the use of *save* and *restore* switches located on the tape drive itself. While these switches do provide full-volume image backup capability, they are intended for service-personnel usage only.

The second method is "direct" access to the tape with the Pascal File System, a method which can be used for selective backup of files and logical volumes, even those not on a CS80 disc.

CAUTION

THE DC600 TAPE DRIVES ARE INTENDED FOR USE AS STREAMING DEVICES. THUS, USING THESE TAPES FOR DIRECT ACCESS AND SELECTIVE BACK-UP, ALTHOUGH SUPPORTED, MAY CAUSE ACCELERATED WEAR OR DAMAGE TO THE TAPE DRIVE AND TAPE. IN OTHER WORDS, USE THESE TAPES ONLY FOR LIMITED BACK-UP AND EMERGENCY PURPOSES, NOT FOR NORMAL FILE SYSTEM CALLS IN USER PROGRAMS OR AS PART OF A BOOT SEQUENCE.

Using the Tape Backup Utility

The Tape Backup Utility (TAPEBKUP.CODE) is a program that enables you to copy the complete image of a disc onto a tape, or vice-versa. The utility also provides operations for certifying tapes and verifying the readability of either discs or tapes.

It is important to note that the utility only provides for complete image backup; it does not provide for selective file or volume backup. A limited amount of selective backup is available by using the Pascal file system for "direct" access to the tape.

Concepts and Terminology

Single and Dual Controllers: With the CS80 integrated disc/tape products, the standard option is for the disc and tape drives to share a common controller. The disc is unit 0; the tape is unit 1. One of the features of the shared-controller product is its ability to transfer data directly from unit 0 to unit 1 or vice-versa, without having the data travel through the host computer. This utility was written specifically to support this mode of operation, as it is the most efficient method for complete backup.

There is an option for the integrated disc/tape products where the disc and tape drives each have their own dedicated controller. Each controller has a separate HP-IB port and bus address, and no logical association with the other one. As a result, the "switch" backup capability is not available with this option. Likewise, *this utility does not support the dual controller option.*

Source and Destination Mis-matches: To be consistant with the product's built-in "switch" backup capability, the utility's Medium-copy operation allows all combinations of source and destination sizes, even those which might seem illogical. Thus, if you have more than one of the disc drives in this family, be sure to mark the type of drive which is backed up on each tape. For instance, if you were to restore a tape backup of a 7908 onto a 7911, much of the 7911 would be inaccessible until you re-Zero'ed it appropriately.

Tape Certification: Tape certification is a procedure very similar to hard disc initialization. Even though the tape comes pre-formatted from the manufacturer, it needs thorough testing, with a possible sparing of bad blocks, before it is ready for use. The addresses of spared blocks are entered into a sparing table and those blocks are never used again. While the tape certification process is somewhat lengthy, tapes usually need to be certified only once during their lifetime. Tapes can be purchased that are already certified.

Tape Auto-sparing: Any time problems occur in the reading of a tape block, the tape controller will record this fact on the tape's permanent log, and then automatically spare out the troublesome block during the next write operation to it. This way, the tape actually tends to get better with usage; slightly marginal blocks that may have escaped detection during certification can be spared later. Note, however, that if a tape is re-certified, the previous sparing information is lost, and all defective blocks will have to be re-discovered.

The utility may in certain instances print "Tape certification in progress", and then almost immediately print "Tape certification completed". In this case, the tape was determined to already be certified, so it was *not* re-certified; it merely went through an optimization of its sparing tables.

Tape Unload Sequence: A loaded tape must go through a logical unload sequence before the tape drive will allow you to physically eject it. A tape unload sequence can be *initiated* either by the front panel *UNLOAD* switch or by the utility. Either way, the tape will then go busy for some period of time, to position it for unloading and to update its permanent logs. A minute or two later, you may hear a buzzing noise made by the tape drive heads as the sequence completes and the busy light extinguishes. You may now physically eject the tape.

When the utility prints "Tape unload request completed", it means that the request to the tape drive to *initiate* the unload sequence has completed. You will have to wait for the unload sequence itself to complete before you will be able to eject the tape.

Verification: Verification is a read without the transmission of data back to the host. The device still does its internal data integrity checks, although it usually inhibits the automatic retry mechanism employed by normal reads. In a verify, the data is not actually compared to anything; the device merely verifies that it can read the data correctly.

To Verify or Not Verify a Tape: Explicit verification of a tape takes as much time to do as normal reads or writes. Thus, in deciding whether to verify or not, you must weigh the time it takes to do the verify versus the extra assurance provided by it. With the present series of integrated disc/tape drives (i.e.,7908, 7911, 7912, and 7914), tape verification *is* recommended.

With the stand-alone HP 9144 Tape Drive, however, the drive incorporates a special read-after-write head, which allows verification of the readability of the data *as it is being written.* With these drives, explicit verification is *not* recommended, although it can still be performed.

If a Disc Doesn't Verify: If a disc gives trouble verifying, the recommended procedure is to save its contents to a tape if desired, then re-initialize it using MEDIAINIT.CODE found on the ACCESS: disc. MEDIAINIT will perform a two-pass error rate test on the entire disc, and then intensively test further any blocks with which the disc controller "remembers" having had trouble. All bad blocks will be spared. After MEDIAINIT completes, the saved contents of the disc can be restored from the tape if need be.

In performing a save of the contents of the troublesome disc mentioned above, the utility may report bad blocks on the source, although not necessarily, since a verify inhibits read retries while a copy does not. In such a case, a best guess of the bad blocks' data would be sent to the tape, and the copy operation would complete. The tape would now contain one or more blocks with corrupted data, but it would "verify" correctly, assuming that it and the tape drive were good. Likewise, after restoring the data back to the freshly-initialized disc, the disc would have the tape's identically corrupted data, and it too would now "verify" correctly.

If a Tape Doesn't Verify: If a tape gives trouble verifying, *do not re-certify it*, merely repeat the write operation to the tape again. The utility always uses the tape in auto-sparing mode.

Specifics on 7914 Backup: To be consistant and fully compatible with the 7914's "switch" backup behavior, the utility

- Always requests two tapes
- Doesn't complain if they are not long tapes
- Doesn't complain if the two tapes do not correspond to each other

Even though rigerous checking is not provided, if you exercise moderate caution you shouldn't have any problems.

With a save, the utility always writes the "first half" tape first, followed by the "second half" tape. With a restore, the utility allows you to insert the tapes in either order; an internal "copy start address" field on the tape specifies which area to the disc to restore it to. The utility also prints out the source and destination start addresses for each copy segment, so that you can detect it if you accidentally restore two "first half" tapes or two "second half" tapes.

How to Invoke the Utility: The utility is quite simple to use. Its user interface is similar to the other Pascal subsystems. The `TAPEBKUP.CODE` utility is delivered on the ACCESS: disc. Like any other program, have the code file on-line and use the eXecute command from the Main Command Level to run the utility. When prompted: "Execute what file ?", type:

`ACCESS:TAPEBKUP` Return or ENTER

The following prompt appears on your CRT.

```
Tapebkup: Medium-copy Verify Certify-tape Quit ?
```

Typing the appropriate letter M, V, C, or Q selects the corresponding operation.

The Medium-copy Operation

The Medium-copy operation prompts for source and destination media. You specify the source media by entering the volume specification of one of the logical volumes on the media. For instance, `#11:` is often first logical volume on a multi-volume hard disc. After one of the disc volumes has been specified, you are shown a listing of all the other logical volumes that will be affected. The specification for the tape media is typically `#41:`

Medium-copy confirms that you have not specified the same media twice, and that the two associated drives are on a shared controller. If not, it aborts the operation.

Medium-copy also checks the medium sizes and gives one of two informative messages for the situations where the destination is a tape, and the tape is not large enough to hold the entire source image. If the source is a 7914 disc, in which case the only method of complete backup is with *two long* tapes and appropriate swapping, you are reminded of the fact. If the source is not a 7914 disc, in which case a complete backup *cannot* be performed, you are advised of this situation, one which you should normally avoid!

At this point, the utility will ask:

```
Are you SURE you want to proceed? (Y/N)
```

Confirm your selections, and respond with Y or N.

If the destination is a tape, you are given the option to automatically verify it after the copy completes. As usual, respond with Y or N.

If the destination is the tape and it has never been certified, it will now go through that process. Tapes must be certified in order to support auto-sparing. Note that the "switch" save operation does not automatically certify tapes before writing to them.

The copy now takes place under control of the device itself. It proceeds at a rate of about 35 Kbytes per second, or roughly two Megabytes per minute. At this rate, copies with a 7908 take about eight minutes, a 7911: about 14 minutes, a 7912: a little over 30 minutes, and a 7914: also a little over 30 minutes per tape, or about 65 minutes total. All errors are reported to the CRT. If the destination is a tape and it is not completely filled by the copy, an end-of-file mark is appended to the valid data.

If the destination is a tape and you opted for auto-verification, the verify occurs at this point. Only the data actually written to the tape is verified, so that time will not be consumed verifying the entire tape if data was copied to only a fraction of it.

The utility allows you several options if some error occurs in the above certify/copy/verify segment. This is primarily motivated by the 7914's two tape backup sequence, but it is also a nice feature for the single tape sequences. Specifically, if an error does occur, you may elect to:

- Retry the same segment on the same tape
- Manually change tapes and retry the same segment with a different, supposably better tape
- Ignore the error and proceed, usually to the next segment of a 7914 two-tape sequence
- Abort the entire sequence

Once a segment attempt has been completed, either because there were no errors or because you elected to ignore them, the utility automatically *initiates* the tape unload sequence. If you have a 7914, the utility then prompts you to change tapes, and proceeds with the second tape's certify/copy/verify segment.

Finally, if the destination is a disc, an automatic full-volume verification is performed.

The Verify Operation

The Verify operation prompts for a media specification, which may be either tape or disc. Like the Media-copy operation, you specify the media by giving the volume ID for one of the volumes on the media. The utility then prints out all associated volumes, and asks for confirmation to proceed. Type Y or N.

If the device is a tape, you are also given the option for the utility to automatically initiate the tape unload sequence after the verify. Respond with Y or N.

The verify performed here always covers the entire medium, even if the medium is a tape with file marks embedded in it. In contrast, the optional verify of a destination tape during the Medium-copy operation verifies only the data just copied to the tape.

As the verify proceeds, the addresses of all unreadable blocks are printed to the screen. The verify is considered to have failed if any are encountered.

If you requested the auto-unload option for a tape, and the verify fails, the utility will *not* unload the tape, in anticipation that you will want to take further action with the tape.

The Certify-tape Operation

Providing this operation separately may seem unnecessary since the Medium-copy operation automatically certifies uncertified tapes before it writes to them. However, it has been included in the utility in case you want to certify one or more tapes without having to copy a disc image to each one at this time.

Another use of this operation is to force re-certification of previously-certified tapes. You would want to do this only if you suspect that blocks on the tape had somehow been spared when they were really OK. This might have happened on a tape drive with dirty heads.

The Certify operation prompts for media specification, confirmation of your choice, and the tape auto-unload option, in the same manner as with the Verify operation. In addition it asks:

```
    Re-certify if already certified? (Y/N)
```

Normally, you will want to type (N), so that that certification will be done only if the tape has never been certified before. However, if you really want to force a re-certification of the tape, with the resultant loss of any previous sparing information, type (Y).

Quitting the Utility

Simply terminates the utility program.

Using the File System for Direct Tape Access

Pascal 3.0 does provide you with the capability of directly accessing the tape like you would with any other mass storage device. If one DC600 tape drive is present, it will be assigned as a single LIF volume, unit #41; a second drive will be assigned #42:. The intent of this capability is to allow you to initialize the tape using MEDIAINIT, then using the Filer, transfer files or volume images to it, list its directory, change its volume name, etc.

You can also use the File System to access the *first* volume of a multi-volume disc image that has been backed-up on tape using TAPEBACKUP. You will not be able to access subsequent logical volumes, nor in general access the second tape of a 7914 backup, without first restoring the image to the disc.

CAUTION

THE DC600 TAPE DRIVES ARE INTENDED FOR USE AS STREAMING DEVICES. THUS, USING THESE TAPES FOR DIRECT ACCESS AND SELECTIVE BACK-UP, ALTHOUGH SUPPORTED, MAY CAUSE ACCELERATED WEAR OR DAMAGE TO THE TAPE DRIVE AND TAPE. IN OTHER WORDS, USE THESE TAPES ONLY FOR LIMITED BACK-UP AND EMERGENCY PURPOSES, NOT FOR NORMAL FILE SYSTEM CALLS IN USER PROGRAMS OR AS PART OF A BOOT SEQUENCE.

When you want to use the tape for selective backup/retrieval versus complete backup/retrieval, you have to be careful how you do it, in order to avoid a couple of common pitfalls. These pitfalls are associated with the inherent characteristics of a streaming tape drive, namely its slow seek times and its inability to start and stop rapidly.

For each file written to the tape the following sequence occurs:

1. A seek is performed to the very beginning of the tape to scan the directory
2. The entire directory is scanned, one block at a time
3. A seek is performed somewhere "out in the middle" of the tape to write out the file body
4. A seek is performed back to the beginning of the tape to update the directory

With this information at hand we now discuss two general rules.

Avoid Large Directories on the Tape

Considering that streaming tapes like these can't stop and start between blocks, but actually coast to a stop, back up, and take a running start at the next block, you can see that scanning a large directory one block at a time will be a painfully slow process. In addition, it accelerates wear on both the tape and the tape drive.

What constitutes a "large" directory? You'll ultimately have to decide, but the following data should aid you in making your decision. On a tape with a LIF directory, the first block will contain the LIF volume label and sixteen directory entries. Each block thereafter can contain thirty-two directory entries. Thus, the logical breakpoints in directry sizes are 16, 48, 80, ... 16 + 32N. MEDIAINIT and the Filer's zero command default to 80 directory entries; it is generally recommended that you not go above this size.

Avoid Transferring Numerous Small Files

Considering that each seek on the tape may take up to tens of seconds, you can see that if you transfer numerous small files, you will probably spend a high percentage of your time seeking back and forth on the tape, and a very small percentage of your time actually transferring data.

Volume Backup

An excellent way to efficiently backup numerous small files is to keep all of them on a single logical volume of the disc, and then backup the entire logical volume in one operation. Two previously seldom-used capabilities of the Filer are volume-to-file and file-to-volume transfers; they provide the key mechanism.

A volume-to-file transfer uses an entire logical volume as the source and saves its complete image as a single file on the destination volume. For example, from the Filer you type (F) to specify a file copy, then type:

```
V11:,#41:VOLBACKUP (Return) or (ENTER)
```

to save the entire image of V11: to a file named VOLBACKUP on volume #41, the tape. While a volume image is in a file, the files within the volume are inaccessible, at least to the average user. To make the files within the volume accessible again, you have to transfer the volume image back to a suitable volume, which is usually the one it originally came from, but need not be as long as it has enough room.

A file-to-volume transfer uses a single file as the source and restores it as a logical volume on the destination volume. For example, from the Filer you type (F) to specify a file copy, then type:

```
#41:VOLBACKUP,#11: (Return) or (ENTER)
```

to restore the file VOLBACKUP, which we assume is a volume image, to its original place. Note thet whatever was on #11 is about to be completely overwritten, so the Filer warns you of this, and asks for your confirmation before proceeding.

Advantages to Selective Backup and Retrieval

Even considering the known pitfalls, selective backup/retrieval to the tape with the Filer is an extremely valuable capability. Here are some advantages:

- You can backup only the files/volumes which changed since the last backup, possibly saving time and the amount of media required for backup.
- You can use the CS80 tape to backup files/volumes from *any* and *all* Pascal-supported mass storage devices, and not just the associated CS80 disc.
- You can interchange data with other HP machines that support LIF on DC600 tapes.
- A single tape can hold may many revisions of the same file/volume, for instance during program development. All revisions of the file/volume must be named uniquely, of course.

Technical Reference

This appendix contains the following useful reference information.

- A "System History" section that describes the additional features provided by Pascal System versions 2.x and 3.0
- A discussion of file interchange between the Pascal System and Series 200 BASIC Systems
- A list of module names used by this Operating System
- A physical memory map
- A software memory map

System History

This section first briefly describes the 1.0 version Pascal Workstation System, and then describes each subsequent version from the standpoint of what features have been added or changed by the version. It is intended to help you make the transition from earlier versions of the system to the 3.0 system.

Pascal 1.0

Here is a brief description of the Pascal 1.0 System. It is put here in order to give you a reference point from which to begin the comparison of later systems.

System Discs

The Pascal 1.0 Workstation System was distributed on a set of four mini-floppy discs, plus one additional disc for documentation. Here are the disc names:

```
BOOT:
SYSVOL:
ACCESS:
COMPASM:
DOC:
```

The SYSVOL:SYSTEM.LIBRARY file contained the entire complement of IO, GRAPHICS, and INTERFACE modules. The unmodified BOOT:SYSTEM.INITLIB contained device-driver software for all peripheral devices supported by the 1.0 system.

Documentation

Documentation for the 1.0 system included the following five manuals.

Problem Solving and Programming with Pascal – This is the textbook from which you can learn about Pascal programming, if you don't already know how to program in this language.

Pascal Language System User's Manual – This manual described booting the system and using each of the subsystems, such as the Editor, Compiler, and Assembler.

Pascal Procedure Library User's Manual – This manual described using the libraries supplied with the system. The libraries consisted of I/O, graphics, LIF-ASCII Filer, Heap Management, and other procedures (etc.) provided with the system.

MC 68000 User's Manual – This manual described MC68000 processor hardware and instruction set.

The Pascal Handbook – This manual described the Pascal language and extensions supported by the Series 200 Computers.

Computers Supported by 1.0

Pascal 1.0 supported only the 9826 and 9836, since they were the only Series 200 computers in production at the introduction of the Pascal 1.0 Workstation System.

Peripheral Devices Supported by 1.0

Pascal 1.0 supported the following mass storage devices:

- Internal 5.25-inch flexible disc drive
- HP 9885 and 9895 8-inch Flexible Disc Drives
- HP 9134 Hard Disc Drives

Pascal 2.0 and 2.1

Here are the additions to the 1.0 system and differences between the 2.1, 2.0, and 1.0 versions of the system.

System Discs

Pascal 2.0 and 2.1 Systems were distributed on six system discs, plus one documentation disc. Here are the names of the discs.

```
BOOT:
SYSVOL:
ACCESS:
CMPASM:
LIB:
CONFIG:
DOC:
```

In contrast to the 1.0 file, the 2.x SYSVOL:LIBRARY was almost empty; the IO, GRAPHICS, and INTERFACE libraries were supplied on separate discs. The user could put just the ones he wanted into his System Library (usually the LIBRARY file). In addition, the Pascal 2.1 GRAPHICS library was re-structured internally and at the user-procedure level.

The Initialization Library (BOOT:INITLIB) supplied contained device-driver software for the most common peripherals but not for all; this was done to conserve memory for the average user, since Pascal 2.x supported many more peripheral devices. The less commonly needed drivers were supplied on the separate CONFIG: disc. Thus, to configure a system to use certain peripherals, the Librarian needed to be used to install the required driver software in INITLIB. Documentation was provided which explained how and when to add optional modules to the INITLIB file.

Documentation

Documentation for the 2.0 system consisted of the five manuals supplied with the 1.0 system, plus the additional *System Internals Documentation* set. This set consisted of these three manuals:

Pascal 2.0 System Designer's Guide – This manual described much of the inner workings of the Pascal system. It contained enough detail to allow you to use many of the "kernel" modules, and it also provided a fairly detailed description of Boot ROM contents and internal computer (hardware and software) architecture.

Pascal 2.0 Source Code Listings (Volume I) – This manual consisted of a cross reference of Pascal procedure names used in the system, and listings of Assembler language modules in the system.

Pascal 2.0 Source Code Listings (Volume II) – This manual consisted of the listings of many Pascal modules used in the system.

File System

HP's Logical Interchange Format (LIF) directory structure was made the primary disc organization for 2.0 and later versions. (LIF ASCII files are intended for interchangability with other HP products.) The 1.0 file system was only able to cleanly handle UCSD directory organizations. HP provided a library of routines to access LIF discs, but they were not integrated into the File System.

The LIF library is not present in the 2.0 and later versions, since it is no longer necessary. The Lfiler (LIF Filer) is also unnecessary and has gone away, since the standard system Filer can now do the job. The 2.0 and later Filers are completely revised programs, although their behaviors are as similar as possible to the 1.0 Filer.

If you were using the 1.0 version and are switching to a later release, don't panic! This does not mean that Pascal 1.0 discs are inaccessible, or even that you need to convert them. See the Special Configurations section of the Technical Reference Appendix for details.

The 2.0 and later File Systems are completely reorganized in comparison to the 1.0 File System. The File System is now broken into levels called File Support (FS), Directory Access Method (DAM), Access Method (AM), and Transfer Method (TM). This organization allows the system to handle any number of different directory formats, and separates out the processing of each type of file structure which is supported. In fact, a customer can invent a new directory format or file type and bind it into the system so it can be used by all programs.

The Directory Access Methods now supported are as follows:

- HP Logical Interchange Format (LIF)
- Shared Resource Manager hierarchical "structured" format (SDF)
- UCSD-compatible (same format as Pascal 1.0)

All these directory organizations are available through normal Pascal file operations. Files generated under Pascal 1.0 are all still fully usable. However, the newer systems can generate files and discs which cannot be properly interpreted by the 1.0 File System.

System File Names

The names of system files were changed with the 2.0 system. They were changed because their length was longer than allowed by the LIF directory format. The name changes are as follows:

Old 1.0 File Name	**New Name**
SYSTEM.LINKER	LIBRARIAN
SYSTEM.EDITOR	EDITOR
SYSTEM.FILER	FILER
SYSTEM.COMPILER	COMPILER
SYSTEM.ASSMBLER	ASSEMBLER
SYSTEM.LIBRARY	LIBRARY
SYSTEM.TABLE	TABLE
SYSTEM.INITLIB	INITLIB
SYSTEM.MISCINFO	MISCINFO
SYSTEM.STARTUP	STARTUP

Object Code Compatibility

Several internal File System changes were made with Pascal 2.0. These changes resulted in corresponding changes in the internal representation of object code files. In general, when a version of the system is not compatible with other versions, the leading digit of the version number will be changed. For instance, versions 2.0 and 1.0 are not object-code compatible, while versions 2.1 and 2.0 are.

While it is regrettable, there really is no alternative to these compatibility restrictions. On the positive side, Pascal application programs which don't "fiddle around" in the operating system are forward compatible to 2.x, so recompilation is all that's necessary.

Supervisor Vs. User State

In versions 2.0 and later, user programs run in the 68000's "user" privilege mode, using the user stack pointer (USP). Interrupts run in "supervisor" privilege mode, using the system stack pointer (SSP). This has implications for calling Boot ROM routines, etc. See the *MC60000 User's Manual* for further details regarding these states.

Additional Computer Supported by 2.0

- Model 16 (HP 9816)

Additional Computer Supported by 2.1

- Model 20 (HP 9920)

Peripheral Configuration

The Pascal 2.x BOOT:TABLE auto-configuration program scanned interfaces for various peripherals and automatically assigned File System unit numbers to devices found (if possible). That was a considerable improvement over the 1.0 version of TABLE.

A source-code version (CTABLE.TEXT) was provided with the system. You could look at the program and read the corresponding commentary in the Special Configurations section of the Technical Reference Appendix to see exactly how the auto-configuration program works. You could also modify certain portions of it to make your own special configurations.

Additional Peripherals Supported by 2.0

Here are changes to the list of disc peripherals supported by Pascal 2.0.

- The CS/80 discs (7908 family)
- The Shared Resource Management system
- The HP 8920x 5.25-inch Flexible Disc Drives
- The HP 9121 3.5-inch (Single-Sided) Flexible Disc Drives
- Several new versions of the HP 913x Hard Disc Drives (they appear as one large volume instead of four smaller ones)
- Certain less obvious features were also added. For instance, the 2.0 system could be fairly easily configured to run from a terminal instead of the built-in CRT and keyboard.

Miscellaneous

Up to 65 Kbytes of Global space has been made available with 2.0 and later versions. This change involved a redefinition of the use of register A5, which now points to an address 32 Kbytes *below* the start of Globals rather than above the first global variable. Consequently, routines in the Boot ROM cannot any longer be called directly; a small interfacing routine is now required to set up the registers and fool the TRY-RECOVER mechanism when calling Boot ROM routines.

Pascal 3.0

Here are the differences and additional features provided by the 3.0 version of the system.

System Discs

The Pascal 3.0 System is distributed on 8 discs, plus two for documentation. Here are the names of the discs.

```
BOOT:
SYSVOL:
ACCESS:
CMP:
ASM:
LIB:
FLTLIB:
CONFIG:
DOC:
DGLPRG:
```

The BOOT:INITLIB file contains a more complete set of device-driver modules; for instance, it now contains module CS80 so that these discs will be recognized by the standard system. See the Adding Modules to INITLIB section of the Special Configurations chapter for a complete list of modules and descriptions of each.

Note that the Assembler and Compiler were put on separate discs due to size. The CMP: disc contains the Compiler. The ASM: disc contains the DEBUGGER program (formerly in BOOT:INITLIB) and the new REVASM (reverse assembler) module.

There are two versions of the GRAPHICS library. The FLTLIB:FGRAPHICS library contains modules optimized for using the HP 98635 Floating-Point Math card; they were compiled with the $FLOAT_HDW ON$ Compiler option, and use the 98635 card, if present. The LIB:GRAPHICS library uses routines in the REALS operating system module; these routines also access the Floating-Point Math card, if present, but the overhead in calling the routines decreases execution speed. The 98635 card can be used with all Pascal 3.0 programs, as long as the REALS module is installed (via INITLIB, etc.).

The DGLPRG: disc provides magnetic copy of the example programs given in the new *Pascal 3.0 Graphics Techniques* manual.

Documentation

Here are the documents shipped with the Pascal 3.0 system.

Pascal 3.0 User's Guide – This is a new manual that takes you from booting your system through setting up your "environment." It provides a "guided tour" of several subsystems, such as the Editor and Filer. You will see all of the steps required to enter, store, compile, and run a simple Pascal program.

Problem Solving and Programming with Pascal – This is the textbook from which you can learn about Pascal programming, if you don't already know how to program in this language.

Pascal 3.0 Workstation System – This manual describes in detail all of the subsystems, such as the Editor, Filer, and Compiler. It also describes such topics as how the computer configures itself to access File System peripherals and how to add new peripherals. This manual was formerly the *Pascal 2.0 User's Manual.* The "Getting Started" information (Chapter 1 of the former manual) has been moved to the new *Pascal 3.0 User's Guide.* Two new chapters have been added: Special Configurations and Non-Disc Mass Storage.

Pascal 3.0 Procedure Library – This manual was formerly the *Pascal Procedure Library User's Manual.* It is basically the same as the former manual, except for the removal of the LIF Procedures and Graphics chapters (graphics is now covered in its own separate manual), and the addition of the System Devices and Segmentation Procedures chapters.

Pascal 3.0 Graphics Techniques – This manual is an expanded version of the Graphics chapter of the former *Pascal Procedure Library User's Manual.* It provides several useful techniques that you can use in writing Pascal graphics programs.

HP Pascal Language Reference for Series 200 Computers – This manual describes the HP Standard Pascal language, as well as the implementation dependencies of the Workstation Pascal language.

MC 68000 User's Manual – This manual describes MC68000 processor hardware and instruction set. It is the same manual as shipped with the 2.x Pascal systems. It also covers the 68008 and 68010 processors.

System Devices Procedural Interface

The procedural interface to "system devices" (such as the keyboard, clock, screen, etc.) has been modified. These changes will not affect the way the system looks at the level of *standard* HP Pascal procedures. However, if any of your programs use procedures *below* this uppermost level (such as procedures in an operating system module), then you may have to make some changes. See the System Devices chapter of the *Pascal 3.0 Procedure Library* for complete details.

Additional Computers and Hardware Features Supported by 3.0

- Model 217 (HP 9817)
- Model 237 (HP 9837)

Both Model 217 and Model 237 have a new type of keyboard which requires Pascal 3.0. The keyboard model number is the HP 46020, which uses the HP Human-Interface Link (HP-HIL) to communicate with the computer.

Pascal 3.0 also supports an optional, "mouse" input device, which can be connected to the computer through the HP Human Interface Link (HP-HIL). The driver (provided on the CONFIG: disc) supports using the mouse for cursor-movement input in both horizontal and vertical directions; it also defines the buttons on the mouse as (Return) or (ENTER) and (Select) ((EXECUTE)) keys. You also can access the mouse from your own applications programs; see the System Devices chapter of the Pascal Procedure Library manual for details.

Both Models 217 and 237 may also have processor boards with Memory-Management Unit (MMU) hardware; if so, the product numbers have 'U' suffixes (such as HP 9817U and 9837U). If the cache-memory feature is also present, then the MMU hardware increases the execution speed of programs (because the cache-memory feature is automatically enabled by Pascal 3.0).

The Model 237 implements a new type of display hardware: a bit-mapped combined alpha/graphics display with a raster size of 1024 by 768 pixels on a 19-inch diagonal CRT screen.

Additional Peripherals Supported by 3.0

- The new Command Set/'80 (CS80) discs, including the HP 7914, 7933, and 7935 Disc Drives
- New stand-alone DC600 (CS80) Tape Drives; right now this category only includes the HP 9144 Tape Drive
- New Sub-Set/'80 (SS80) floppy discs; right now this category only includes the HP 9122 3.5-inch Double-Sided Floppy discs
- Several new versions of the 913x Hard Discs (V and XV suffix drives)

Additional Cards Supported by 3.0

Here are the new cards that are supported by Pascal 3.0

- HP 98255 EPROM and HP 98253 EPROM Programmer cards, which can be used as mass storage devices (see the Non-Disc Mass Storage chapter of this manual for details)
- HP 98259 Magnetic Bubble Memory cards, which can also be used as mass storage devices (see the Non-Disc Mass Storage chapter of this manual for details)
- HP 98635 Floating-Point Math card (see the description of the FLOAT_HDW Compiler option in the Compiler chapter for details)
- HP 98257 1-Megabyte Memory card, which supports parity-checking hardware

Peripheral Configuration

How the system boots and auto-configures itself is fully discussed in the Special Configurations chapter. The Pascal 3.0 TABLE program has even more capabilities than the 2.x version: it automatically configures up to 3 floppy disc drives (dual or single) and at least the first hard disc in the system (up to 10 are potentially possible).

A source-code version of the 3.0 TABLE program (CONFIG:CTABLE.TEXT) is also provided with the 3.0 system. You can look at the program and read the corresponding commentary in the Special Configurations chapter to see exactly how the auto-configuration process works, and you can modify certain portions of it to make your own special configurations. A major change with the 3.0 TABLE is that now you can "coalesce" logical volumes on hard discs without the need to modify and re-compile the TABLE source program.

The CTABLE program can now easily support printers with RS-232C interfaces by making one small change in the program and re-compiling. See the Special Configurations chapter of this manual for details.

Object Code Compatibility

Several internal changes were made with Pascal 3.0. These changes resulted in corresponding changes in the internal representation of object code files. In general, when a version of the system is not compatible with other versions, the leading digit of the version number will be changed. For instance, versions 3.0 and 2.0 are not object-code compatible, while versions 2.1 and 2.0 are.

While it is regrettable, there really is no alternative to these compatibility restrictions. On the positive side, Pascal application programs which don't "fiddle around" in the operating system are source-code compatible with 3.0, so recompilation is usually all that's necessary.

General System Features Added by 3.0

Stream Files: Stream files on read-only devices are now allowed; adding the `[*]` specifier to the stream file name allows this usage by disabling the prompt feature. This same mechanism also allows the use of a stream file called AUTOKEYS to provide "autostart" capabilities with read-only system volumes. See the description of the Stream command in the Overview chapter of this manual for details.

Filer: The Filer can now perform a Translate operation to the CONSOLE: volume, with the ability to view the translated file one screen at a time. See the Filer chapter of this manual for details.

Compiler: The following Compiler options were added. The WARN option allows you to disable warning messages. The FLT_HDW option allows you to specify one of three actions: ON specifies that the Compiler is to emit code that assumes a 98635 Floating-Point Math card is installed in the computer; TEST specifies that the emitted code is to test for the presence of the card; OFF specifies that emitted code always uses floating-point library routines. See the Compiler chapter of this manual for details.

Assembler: The Assembler has been modified to allow use of the new op codes provided by the 680xx processors (such as the MOVES and RTD instructions).

Librarian: A special "edit" mode was added to the Librarian. It allows you to add modules to an existing library more easily. The Librarian can also unassemble the new instructions for the 680xx processors (such as the MOVES and RTD instructions). See the Librarian chapter of this manual for details.

Debugger: These are the new commands that have been added to the Debugger: "X" format for reverse assembly; "R" format for displaying REAL numbers; "O" format and FO default format for octal numbers; added repeat counts on format specifiers; "!" input format for binary numbers; "%" input format for octal numbers; relational operators can now be used in expressions; DA and DG commands for DUMP ALPHA and DUMP GRAPHICS functions, and also the ability to use the corresponding keys; four more softkeys now available (10 total); five more breakpoints now available (9 total); PN and PX commands (PX is an alternate syntax for the existing P command); IF, ELSE, and END commands for conditional execution of Debugger commands added; CALL command added; EC and ETC commands added; (PAUSE) key definition changed. See the Debugger chapter of this manual for details.

Segmentation Procedures: Several procedures that add the capability of run-time program segmentation have been added to the system. See the Segmentation Procedures chapter of the *Pascal 3.0 Procedure Library* manual for details.

File Interchange Between Pascal and BASIC

You may wish to exchange data on file between the Pascal and BASIC environments. There are a few rules you should follow.

- Pascal and BASIC treat LIF directories on flexible discs similarly. ASCII text files are intended to be used as the interchange mechanism.
- It was mentioned earlier that Pascal compresses the suffix of user file names in order to effectively allow longer file names. BASIC doesn't know about compressed names, so the BASIC program needs to invert the compression algorithm. This inversion is very simple, and is described in the section of the File System chapter called Programming with Files. Essentially, Pascal chops off the dot and the suffix (such as .ASC), then appends the first letter of the suffix and enough trailing "_" characters to make a 10-character name. Thus "ABC.ASC" becomes "ABCA______", which is the name BASIC will see.
- BASIC can't deal with more than one LIF directory on a hard disc; Pascal, unless told otherwise, wants to divide large hard discs into several volumes each with its own directory. Hard disc partitioning is described in the Special Configurations chapter.

If a disc is initialized by BASIC, Pascal and BASIC will both see the disc as one very large volume. Pascal's preference to partition the disc is overridden by what BASIC actually did.

If a disc is initialized by Pascal and partitioned into multiple volumes, BASIC will only see the first volume and will not be able to access any part of the disc beyond the first volume. Pascal will see all the volumes.

See the Special Configurations chapter for information on forcing Pascal to treat a partitionable disc as a single volume.

Module Names Used by the Operating System

Here are the names of modules that are present in the system as it is shipped from the factory. They are provided so that you will not name a module using any of these names, unless you *definitely* want to override the system module's function.

Note that many of these module names do not show up in the system symbol table (for example, they may have been removed by linking). However, you should not use them, because HP reserves the right to use them in the future.

A complete list of symbol names actually used in the system is provided on the DOC: disc in the file named SYMBOLS.

A804XDVR
A804XINIT
ABORT_IO
ALLREAL_STUFF
ALPHAFLAG
ALPHALIST
AMIGO
AMIGODVR
AMIGOINIT
ASC_AM
ASM
BAT
BKGND
BOOT
BOOTDAMMODULE
BUBBLE
BUBBLES
BUB_DVR
CHECK_TFR
CHECK_TIMER
CHOOK
CI
CLOCK
CMD
CONVERT
CONVERT_TEXT
CRT
CRTB
CS80
CS80DSR
CS80DVR
CS80INIT
CS80_CHAN_INDEP_CLR
CSAMIGO

DATA_COMM
DC_INITIALIZE
DEBUGGER
DELAY_TIMER
DGL_ARAS
DGL_ATEXT
DGL_AUTL
DGL_CONFG_IN
DGL_CONFG_OUT
DGL_GEN
DGL_HPGL
DGL_HPGLI
DGL_IBODY
DGL_INQ
DGL_KNOB
DGL_LIB
DGL_POLY
DGL_RAS
DGL_RASTER
DGL_TOOLS
DGL_TYPES
DGL_VAR
DGL_VARS
DISCHPIB
DISCINT_INITIALIZE
DISC_INTF
DMA
DMA_INITIALIZE
DMA_STBSY
DROPDMA
DRVASM
EDRIVER
EPROMS
ERR_INFO

EVALGVR
EXCP
EXTDC
EXTDL_EDI
EXTG_EG
EXTH_EH
F9885
F9885DVR
F9885INIT
FGATOR
FLTPTHDW
FS
GENERAL_0
GENERAL_1
GENERAL_2
GENERAL_3
GENERAL_4
GETDMA
GLE_ARAS_OUT
GLE_ASCLIP
GLE_ASTEXT
GLE_AUTL
GLE_FILE_IO
GLE_GEN
GLE_GENI
GLE_HPGL_IN
GLE_HPGL_OUT
GLE_HPIB_IO
GLE_KNOB_IN
GLE_RAS_OUT
GLE_SCLIP
GLE_SMARK
GLE_STEXT
GLE_STROKE

GLE_TYPES
GLE_UTLS
GP
GPIO
GPIO_INITIALIZE
GRAPHICSBASE
GRAPHICSFLAG
G_DOLLAR
HPHIL
HPIB
HPI
B_0
HPIB_1
HPIB_2
HPIB_3
HPIB_INITIALIZE
HPM
HV50
HV52
INITBAT
INITBKGND
INITCRT
INITKBD
INITLOAD
INITUNITS
INIT_BKGND
INIT_DC
INIT_DISCINT
INIT_DMA
INIT_GPIO
INIT_HPIB
INIT_RS
INIT_SRM
INIT_TIMER

INSTALL_UCSD_DAM
INSTLIFDAM
INST_EPROM
INTDC
IOCOMASM
IODECLARATIONS
IOLIBRARY_KERNEL
ISR
ITXFR
KATA
KBD
KERNEL
KEYS
LAST
LDR
LIFMODULE
LIF_DAM
LOADER
LOCKMODULE
LOGEOT
LOGINT
M68KTYPE
MATCHDEFEXT
MFS
MINI
MISC
MSYSFLAGS
NONUSKBD1
NONUSKBD2
PRINTER
PRINTSYMS
PRTDVR
RAND
RANDOM
REALS
RELOCATE
RESETX
REVASM
REVASM_MOD
RND
ROMAN
RS
RS232
RS_INITIALIZE
SEGMENTER
SERIAL_0
SERIAL_3
SETUPSYS
SRM
SRMAMMODULE
SRMDAMMODULE
STACKFUDGE
STBSY
STCLR
SWAP8
SWAPK
SYSDEVICES
SYSDEVS
SYSGLOBALS
TAPEBUF
TESTDMA
TEXT_AM
TIMED_OUT
TIMEREXISTS
UCSDMODULE
UCSD_AM
UIO
WAIT_TFR
WS1.0_DAM

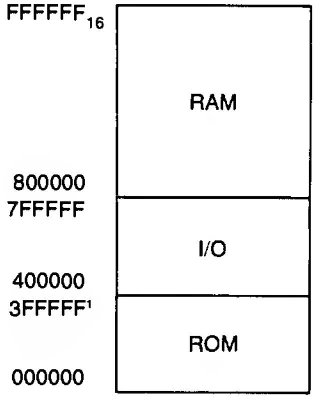
Physical Memory Map

The first part of this section describes the physical hardware memory map of your Series 200 computer with regard to ROM space, I/O space and RAM space. This section begins with an overview of the hardware memory layout, followed by a more detailed memory map of each major section of memory.

Register addresses and descriptions are included for the internal I/O devices.

Full 16 Megabyte Addressing Range

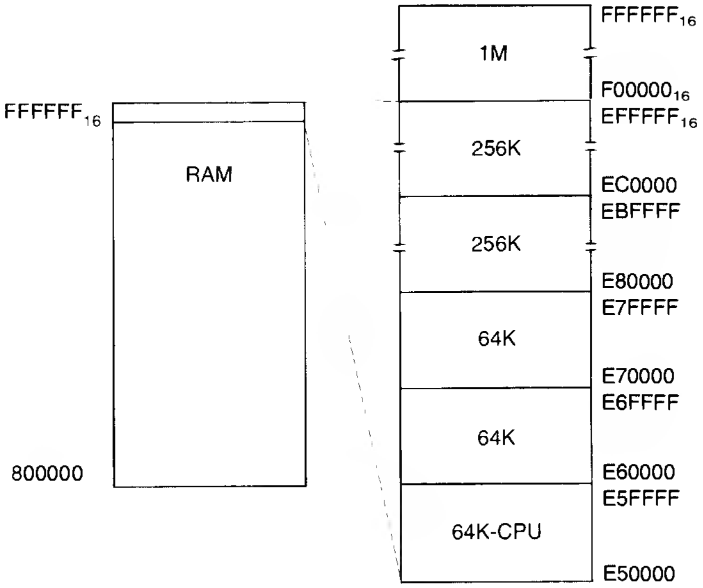
There are 23 address lines (BA1..BA23) providing 16 megabyte addressing on WORD boundaries. For byte operations, two control lines BUDS (Buffered Upper Data Strobe) and BLDS (Buffered Lower Data Strobe) indicate whether the upper data byte (BD8 through BD15), the lower data byte (BD0 through BD7), or both bytes are involved in the communication. Note: When BA0 = 0, the high byte is requested. When BA0 = 1, the lower byte is requested.



[1] On machines with Memory Management Units, the upper limit of ROM address is $1FFFFF.

A Hypothetical RAM Configuration

RAM is located in the top half of the 16 megabyte addressing range. The bank select switches on the boards must be set so that the boards are mapped contiguously from highest memory progressing down. Any 1M boards should be set to the highest memory addresses. Any 256K boards should be mapped in next. Any 64K boards are mapped below those. The 64K on the CPU board automatically maps into the first available address below the plug-in boards.



The 1M memory boards have a 4-bit, user-changeable, bank-select switch. The 4 bits of the switch correspond to the 4 MSBs of the address bus. For example, the 1M board above has its switches set as follows:

```
Msb   Lsb
  1111
```

The 256K memory boards have a 6-bit, user-changeable, bank-select switch. The 6 bits of the switch correspond to the 6 MSBs of the address bus. For example, the two 256K boards above have their switches set:

```
Msb      Lsb
  111011
  111010
```

respectively.

After all the 256K memory boards have been set in high memory, the 64K boards should then be mapped in. It is necessary to set 8 switches, corresponding to the 8 MSBs of the address bus. For example two 64K boards mapped under the 256K boards above would have their switches set:

```
Msb        Lsb
  11100111
  11100110
```

respectively. The 64K on the CPU board automatically maps into the first available memory space down:

```
Msb        Lsb
  11100101
```

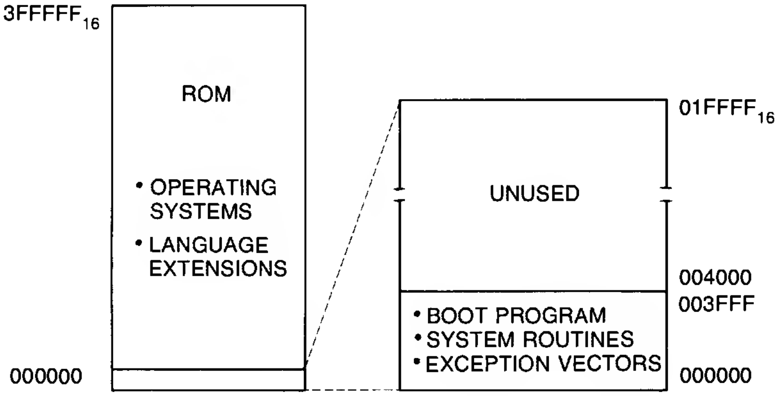
Note

If the memory boards are not mapped contiguously, the auto-locating memory on the CPU board, if any, maps into the first hole left by the memory boards, and those mapped below the hole are not used.

Note

Setting the switches on two memory boards to the same code could cause damage to the memory boards.

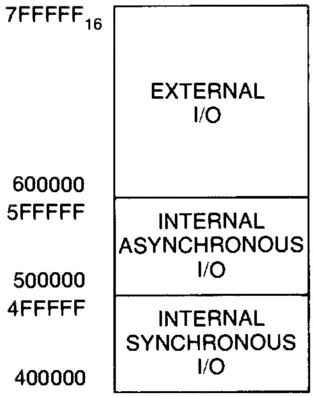
The Overall ROM Memory Map



The boot program, exception vectors, and some system routines reside on ROM chips starting at $000000 and extending to $003FFF. The space between $004000 and $01FFFF is unused. (The BOOTROM 3.0 consists of approximately 48 Kbytes of code, starting at $000000). The boot program checks for system ROMs and language extension ROMs on 16K boundries beginning at $020000 and continuing up to $3FC000. These ROMs are recognized by their appropriate header information.

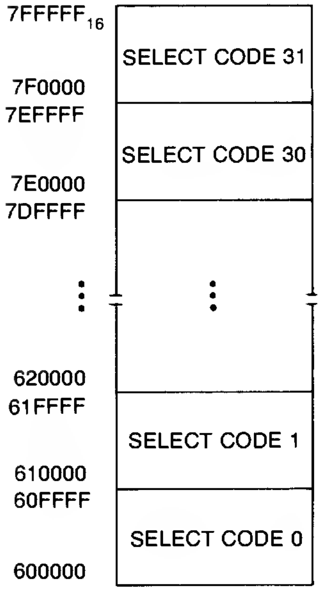
Memory Mapped I/O

I/O memory space is divided into three sections. External I/O is that section which corresponds to the backplane of the Series 200. The select codes on the backplane I/O cards correspond to address bits BA 16 through BA20.

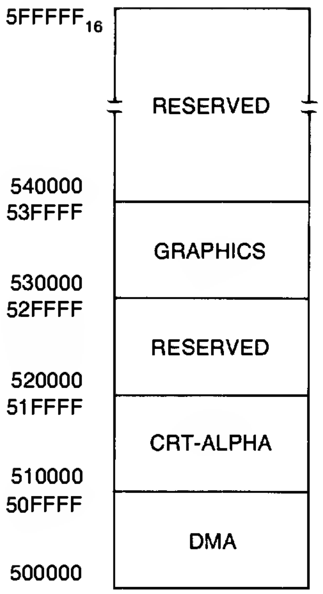


External I/O

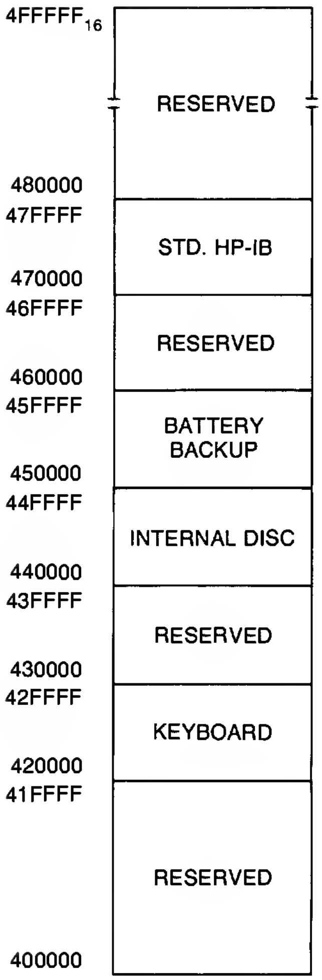
All I/O cards available for the Series 200 are mapped into the external I/O space on 64K boundaries (except the DMA card which maps with synchronous internal I/O). There are 32 such spaces between $600000 and $7FFFFF. User designed cards must also map into one of these spaces. The select codes correspond to address lines BA16 through BA20. HP cards have been assigned default select codes but can be reset by the user to map into any configuration.



Internal I/O



Synchronous



Asynchronous

Internal I/O functions are doubly mapped, once between $400000 and $4FFFFF, and again between $500000 and $5FFFFF. That is why much of the internal I/O space is reserved. Those I/O functions can be addressed in either range. The difference is that the low range generates a DTACK (Data Acknowledge) automatically, whether the card has actually responded to the data or not. If the synchronous cards are addressed in the high range, there will be no DTACK generated at all.

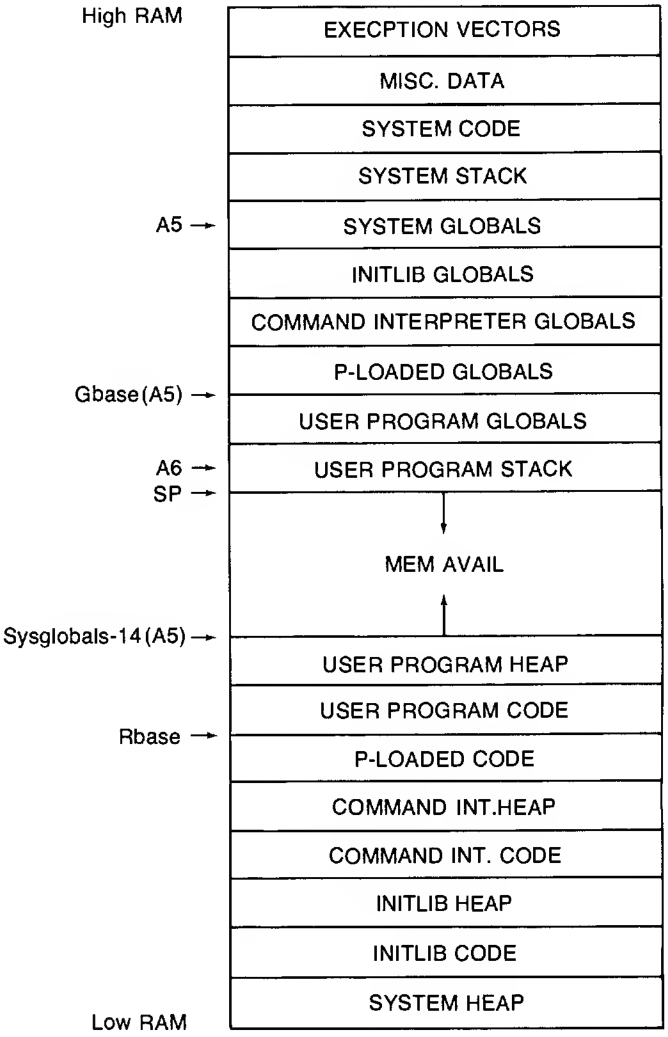
The Software Memory Map

This software memory map shows the symbolic locations of global variables, local variables, the stack, the heap and the code relocation base.

If you'll look at the directory of the boot disk, you'll see the files that are loaded into RAM at power-up. SYSTEM is loaded first. It shows as SYSTEM CODE on the map. It sets up the EXCEPTION VECTORS, MISCelaneous DATA, SYSTEM GLOBALS, SYSTEM STACK, INITLIB GLOBALS and SYSTEM HEAP. TABLE is loaded and it modifies some of the data in SYSTEM GLOBALS and SYSTEM HEAP.

INITLIB is then loaded. It contains the I/O drivers and the Debugger.

Finally, STARTUP is loaded. When you receive a Pascal Language System from HP, STARTUP is the Main Command Level command interpreter. It handles P-loading files, loading subsystem files, and loading user programs. You may use the Filer to change any file you want to STARTUP and that file will execute at power-up. A copy of the BOOT disc should be created before the change is made because you won't be able to change the file back to the original configuration.



Character Sets

This section provides the tables for the following character sets:

- U.S. ASCII character set
- U.S./European display characters (for Models 216, 220, 226, and 236 Computers)
- U.S./European display characters (for Models 217 and 237 Computers)
- Katakana display characters (for all Series 200 Computers)
- CRT highlight characters (for the Model 236 Computer)

U.S. ASCII Character Set

ASCII	EQUIVALENT FORMS				HP-IB
Char.	Dec	Binary	Oct	Hex	
NUL	0	00000000	000	00	
SOH	1	00000001	001	01	GTL
STX	2	00000010	002	02	
ETX	3	00000011	003	03	
EOT	4	00000100	004	04	SDC
ENQ	5	00000101	005	05	PPC
ACK	6	00000110	006	06	
BEL	7	00000111	007	07	
BS	8	00001000	010	08	GET
HT	9	00001001	011	09	TCT
LF	10	00001010	012	0A	
VT	11	00001011	013	0B	
FF	12	00001100	014	0C	
CR	13	00001101	015	0D	
SO	14	00001110	016	0E	
SI	15	00001111	017	0F	
DLE	16	00010000	020	10	
DC1	17	00010001	021	11	LLO
DC2	18	00010010	022	12	
DC3	19	00010011	023	13	
DC4	20	00010100	024	14	DCL
NAK	21	00010101	025	15	PPU
SYNC	22	00010110	026	16	
ETB	23	00010111	027	17	
CAN	24	00011000	030	18	SPE
EM	25	00011001	031	19	SPD
SUB	26	00011010	032	1A	
ESC	27	00011011	033	1B	
FS	28	00011100	034	1C	
GS	29	00011101	035	1D	
RS	30	00011110	036	1E	
US	31	00011111	037	1F	

ASCII	EQUIVALENT FORMS				HP-IB
Char.	Dec	Binary	Oct	Hex	
space	32	00100000	040	20	LA0
!	33	00100001	041	21	LA1
"	34	00100010	042	22	LA2
#	35	00100011	043	23	LA3
$	36	00100100	044	24	LA4
%	37	00100101	045	25	LA5
&	38	00100110	046	26	LA6
'	39	00100111	047	27	LA7
(	40	00101000	050	28	LA8
)	41	00101001	051	29	LA9
*	42	00101010	052	2A	LA10
+	43	00101011	053	2B	LA11
,	44	00101100	054	2C	LA12
–	45	00101101	055	2D	LA13
.	46	00101110	056	2E	LA14
/	47	00101111	057	2F	LA15
0	48	00110000	060	30	LA16
1	49	00110001	061	31	LA17
2	50	00110010	062	32	LA18
3	51	00110011	063	33	LA19
4	52	00110100	064	34	LA20
5	53	00110101	065	35	LA21
6	54	00110110	066	36	LA22
7	55	00110111	067	37	LA23
8	56	00111000	070	38	LA24
9	57	00111001	071	39	LA25
:	58	00111010	072	3A	LA26
;	59	00111011	073	3B	LA27
<	60	00111100	074	3C	LA28
=	61	00111101	075	3D	LA29
>	62	00111110	076	3E	LA30
?	63	00111111	077	3F	UNL

STD-LL-60182

U.S. ASCII Character Set

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
@	64	01000000	100	40	TA0
A	65	01000001	101	41	TA1
B	66	01000010	102	42	TA2
C	67	01000011	103	43	TA3
D	68	01000100	104	44	TA4
E	69	01000101	105	45	TA5
F	70	01000110	106	46	TA6
G	71	01000111	107	47	TA7
H	72	01001000	110	48	TA8
I	73	01001001	111	49	TA9
J	74	01001010	112	4A	TA10
K	75	01001011	113	4B	TA11
L	76	01001100	114	4C	TA12
M	77	01001101	115	4D	TA13
N	78	01001110	116	4E	TA14
O	79	01001111	117	4F	TA15
P	80	01010000	120	50	TA16
Q	81	01010001	121	51	TA17
R	82	01010010	122	52	TA18
S	83	01010011	123	53	TA19
T	84	01010100	124	54	TA20
U	85	01010101	125	55	TA21
V	86	01010110	126	56	TA22
W	87	01010111	127	57	TA23
X	88	01011000	130	58	TA24
Y	89	01011001	131	59	TA25
Z	90	01011010	132	5A	TA26
[	91	01011011	133	5B	TA27
\	92	01011100	134	5C	TA28
]	93	01011101	135	5D	TA29
^	94	01011110	136	5E	TA30
_	95	01011111	137	5F	UNT

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
`	96	01100000	140	60	SC0
a	97	01100001	141	61	SC1
b	98	01100010	142	62	SC2
c	99	01100011	143	63	SC3
d	100	01100100	144	64	SC4
e	101	01100101	145	65	SC5
f	102	01100110	146	66	SC6
g	103	01100111	147	67	SC7
h	104	01101000	150	68	SC8
i	105	01101001	151	69	SC9
j	106	01101010	152	6A	SC10
k	107	01101011	153	6B	SC11
l	108	01101100	154	6C	SC12
m	109	01101101	155	6D	SC13
n	110	01101110	156	6E	SC14
o	111	01101111	157	6F	SC15
p	112	01110000	160	70	SC16
q	113	01110001	161	71	SC17
r	114	01110010	162	72	SC18
s	115	01110011	163	73	SC19
t	116	01110100	164	74	SC20
u	117	01110101	165	75	SC21
v	118	01110110	166	76	SC22
w	119	01110111	167	77	SC23
x	120	01111000	170	78	SC24
y	121	01111001	171	79	SC25
z	122	01111010	172	7A	SC26
{	123	01111011	173	7B	SC27
|	124	01111100	174	7C	SC28
}	125	01111101	175	7D	SC29
~	126	01111110	176	7E	SC30
DEL	127	01111111	177	7F	SC31

U.S./European Display Characters

These characters can be displayed on the alpha screens of Models 216, 220, 226, and 236 Computers.

ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary		Dec	Binary		Dec	Binary		Dec	Binary
NU	0	00000000		32	00100000	@	64	01000000	`	96	01100000
SH	1	00000001	!	33	00100001	A	65	01000001	a	97	01100001
SX	2	00000010	"	34	00100010	B	66	01000010	b	98	01100010
EX	3	00000011	#	35	00100011	C	67	01000011	c	99	01100011
ET	4	00000100	$	36	00100100	D	68	01000100	d	100	01100100
EQ	5	00000101	%	37	00100101	E	69	01000101	e	101	01100101
AK	6	00000110	&	38	00100110	F	70	01000110	f	102	01100110
BL	7	00000111	'	39	00100111	G	71	01000111	g	103	01100111
BS	8	00001000	(	40	00101000	H	72	01001000	h	104	01101000
HT	9	00001001	)	41	00101001	I	73	01001001	i	105	01101001
LF	10	00001010	*	42	00101010	J	74	01001010	j	106	01101010
VT	11	00001011	+	43	00101011	K	75	01001011	k	107	01101011
FF	12	00001100	,	44	00101100	L	76	01001100	l	108	01101100
CR	13	00001101	-	45	00101101	M	77	01001101	m	109	01101101
SO	14	00001110	.	46	00101110	N	78	01001110	n	110	01101110
SI	15	00001111	/	47	00101111	O	79	01001111	o	111	01101111
DL	16	00010000	0	48	00110000	P	80	01010000	p	112	01110000
D1	17	00010001	1	49	00110001	Q	81	01010001	q	113	01110001
D2	18	00010010	2	50	00110010	R	82	01010010	r	114	01110010
D3	19	00010011	3	51	00110011	S	83	01010011	s	115	01110011
D4	20	00010100	4	52	00110100	T	84	01010100	t	116	01110100
NK	21	00010101	5	53	00110101	U	85	01010101	u	117	01110101
SY	22	00010110	6	54	00110110	V	86	01010110	v	118	01110110
EB	23	00010111	7	55	00110111	W	87	01010111	w	119	01110111
CN	24	00011000	8	56	00111000	X	88	01011000	x	120	01111000
EM	25	00011001	9	57	00111001	Y	89	01011001	y	121	01111001
SB	26	00011010	:	58	00111010	Z	90	01011010	z	122	01111010
EC	27	00011011	;	59	00111011	[	91	01011011	{	123	01111011
FS	28	00011100	<	60	00111100	\	92	01011100	|	124	01111100
GS	29	00011101	=	61	00111101	]	93	01011101	}	125	01111101
RS	30	00011110	>	62	00111110	^	94	01011110	~	126	01111110
US	31	00011111	?	63	00111111	_	95	01011111	▒	127	01111111

STD-LL-60182

U.S./European Display Characters

These characters can be displayed on the alpha screens of Models 216, 220, 226, and 236 Computers.

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
NOTE	128	10000000
NOTE	129	10000001
NOTE	130	10000010
NOTE	131	10000011
NOTE	132	10000100
NOTE	133	10000101
NOTE	134	10000110
NOTE	135	10000111
NOTE	136	10001000
NOTE	137	10001001
NOTE	138	10001010
NOTE	139	10001011
NOTE	140	10001100
NOTE	141	10001101
NOTE	142	10001110
NOTE	143	10001111
hp	144	10010000
hp	145	10010001
hp	146	10010010
hp	147	10010011
hp	148	10010100
hp	149	10010101
hp	150	10010110
hp	151	10010111
hp	152	10011000
hp	153	10011001
hp	154	10011010
hp	155	10011011
hp	156	10011100
hp	157	10011101
hp	158	10011110
hp	159	10011111
hp	160	10100000
hp	161	10100001
hp	162	10100010
hp	163	10100011
hp	164	10100100
hp	165	10100101
hp	166	10100110
hp	167	10100111
´	168	10101000
`	169	10101001
^	170	10101010
¨	171	10101011
~	172	10101100
hp	173	10101101
hp	174	10101110
£	175	10101111
¯	176	10110000
hp	177	10110001
hp	178	10110010
°	179	10110011
hp	180	10110100
Ç	181	10110101
Ñ	182	10110110
ñ	183	10110111
¡	184	10111000
¿	185	10111001
¤	186	10111010
£	187	10111011
hp	188	10111100
§	189	10111101
hp	190	10111110
hp	191	10111111
â	192	11000000
ê	193	11000001
ô	194	11000010
û	195	11000011
á	196	11000100
é	197	11000101
ó	198	11000110
ú	199	11000111
à	200	11001000
è	201	11001001
ò	202	11001010
ù	203	11001011
ä	204	11001100
ë	205	11001101
ö	206	11001110
ü	207	11001111
Å	208	11010000
î	209	11010001
Ø	210	11010010
Æ	211	11010011
å	212	11010100
í	213	11010101
ø	214	11010110
æ	215	11010111
Ä	216	11011000
ì	217	11011001
Ö	218	11011010
Ü	219	11011011
É	220	11011100
ï	221	11011101
ß	222	11011110
hp	223	11011111
hp	224	11100000
hp	225	11100001
hp	226	11100010
hp	227	11100011
hp	228	11100100
hp	229	11100101
hp	230	11100110
hp	231	11100111
hp	232	11101000
hp	233	11101001
hp	234	11101010
hp	235	11101011
hp	236	11101100
hp	237	11101101
hp	238	11101110
hp	239	11101111
hp	240	11110000
hp	241	11110001
hp	242	11110010
hp	243	11110011
hp	244	11110100
hp	245	11110101
hp	246	11110110
hp	247	11110111
hp	248	11111000
hp	249	11111001
hp	250	11111010
hp	251	11111011
hp	252	11111100
hp	253	11111101
hp	254	11111110
▒	255	11111111

NOTE: Ignored by the 9826; see 9836 CRT Highlight Characters.

STD-LL-60182

U.S./European Display Characters

These characters can be displayed on the alpha screen of the Model 217 computer and on the bit-mapped graphics screen of the Model 237 computer.

ASCII

Num	Chr	Num.	Chr.	Num	Chr.	Num.	Chr
0	NU	32		64	@	96	`
1	SH	33	!	65	A	97	a
2	SX	34	"	66	B	98	b
3	EX	35	#	67	C	99	c
4	ET	36	$	68	D	100	d
5	EQ	37	%	69	E	101	e
6	AK	38	&	70	F	102	f
7	🔔	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DL	48	0	80	P	112	p
17	D1	49	1	81	Q	113	q
18	D2	50	2	82	R	114	r
19	D3	51	3	83	S	115	s
20	D4	52	4	84	T	116	t
21	NK	53	5	85	U	117	u
22	SY	54	6	86	V	118	v
23	EB	55	7	87	W	119	w
24	CN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SB	58	:	90	Z	122	z
27	EC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	▒

U.S./European Display Characters

These characters can be displayed on the alpha screen of the Model 217 computer and on the bit-mapped graphics screen of the Model 237 computer.

ASCII

Num	Chr.	Num.	Chr.	Num.	Chr	Num.	Chr.
128	$^{C}_{L}$	160		192	â	224	Á
129	$^{I}_{V}$	161	À	193	ê	225	Ã
130	$^{B}_{G}$	162	Â	194	ô	226	ã
131	$^{I}_{B}$	163	È	195	û	227	Ð
132	$^{U}_{\underline{L}}$	164	Ê	196	á	228	ð
133	$^{I}_{\underline{V}}$	165	Ë	197	é	229	Í
134	$^{B}_{\underline{G}}$	166	Î	198	ó	230	Ì
135	$^{I}_{\underline{V}}$	167	Ï	199	ú	231	Ó
136	$^{W}_{H}$	168	´	200	à	232	Ò
137	$^{R}_{D}$	169	`	201	è	233	Õ
138	$^{Y}_{E}$	170	ˆ	202	ò	234	õ
139	$^{G}_{R}$	171	¨	203	ù	235	Š
140	$^{C}_{Y}$	172	˜	204	ä	236	š
141	$^{B}_{U}$	173	Ù	205	ë	237	Ú
142	$^{M}_{G}$	174	Û	206	ö	238	Ÿ
143	$^{B}_{K}$	175	₤	207	ü	239	ÿ
144	$^{9}_{0}$	176	¯	208	Å	240	Þ
145	$^{9}_{1}$	177	$^{B}_{1}$	209	î	241	þ
146	$^{9}_{2}$	178	$^{B}_{2}$	210	Ø	242	$^{F}_{2}$
147	$^{9}_{3}$	179	°	211	Æ	243	$^{F}_{3}$
148	$^{9}_{4}$	180	Ç	212	å	244	$^{F}_{4}$
149	$^{9}_{5}$	181	ç	213	í	245	$^{I}_{O}$
150	$^{9}_{6}$	182	Ñ	214	ø	246	−
151	$^{9}_{7}$	183	ñ	215	æ	247	¼
152	$^{9}_{8}$	184	¡	216	Ä	248	½
153	$^{9}_{9}$	185	¿	217	ì	249	ª
154	$^{9}_{A}$	186	¤	218	Ö	250	º
155	$^{9}_{B}$	187	£	219	Ü	251	«
156	$^{9}_{C}$	188	¥	220	É	252	■
157	$^{9}_{D}$	189	§	221	ï	253	»
158	$^{9}_{E}$	190	ƒ	222	ß	254	±
159	$^{9}_{F}$	191	¢	223	Ô	255	🅺

Katakana Display Characters

These characters can be displayed on all Series 200 Computers (while in Katakana mode).

ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary		Dec	Binary		Dec	Binary		Dec	Binary
␀	0	00000000		32	00100000	@	64	01000000	`	96	01100000
␁	1	00000001	!	33	00100001	A	65	01000001	a	97	01100001
␂	2	00000010	"	34	00100010	B	66	01000010	b	98	01100010
␃	3	00000011	#	35	00100011	C	67	01000011	c	99	01100011
␄	4	00000100	$	36	00100100	D	68	01000100	d	100	01100100
␅	5	00000101	%	37	00100101	E	69	01000101	e	101	01100101
␆	6	00000110	&	38	00100110	F	70	01000110	f	102	01100110
␇	7	00000111	'	39	00100111	G	71	01000111	g	103	01100111
␈	8	00001000	(	40	00101000	H	72	01001000	h	104	01101000
␉	9	00001001	)	41	00101001	I	73	01001001	i	105	01101001
␊	10	00001010	*	42	00101010	J	74	01001010	j	106	01101010
␋	11	00001011	+	43	00101011	K	75	01001011	k	107	01101011
␌	12	00001100	,	44	00101100	L	76	01001100	l	108	01101100
␍	13	00001101	-	45	00101101	M	77	01001101	m	109	01101101
␎	14	00001110	.	46	00101110	N	78	01001110	n	110	01101110
␏	15	00001111	/	47	00101111	O	79	01001111	o	111	01101111
␐	16	00010000	0	48	00110000	P	80	01010000	p	112	01110000
␑	17	00010001	1	49	00110001	Q	81	01010001	q	113	01110001
␒	18	00010010	2	50	00110010	R	82	01010010	r	114	01110010
␓	19	00010011	3	51	00110011	S	83	01010011	s	115	01110011
␔	20	00010100	4	52	00110100	T	84	01010100	t	116	01110100
␕	21	00010101	5	53	00110101	U	85	01010101	u	117	01110101
␖	22	00010110	6	54	00110110	V	86	01010110	v	118	01110110
␗	23	00010111	7	55	00110111	W	87	01010111	w	119	01110111
␘	24	00011000	8	56	00111000	X	88	01011000	x	120	01111000
␙	25	00011001	9	57	00111001	Y	89	01011001	y	121	01111001
␚	26	00011010	:	58	00111010	Z	90	01011010	z	122	01111010
␛	27	00011011	;	59	00111011	[	91	01011011	{	123	01111011
␜	28	00011100	<	60	00111100	¥	92	01011100	|	124	01111100
␝	29	00011101	=	61	00111101	]	93	01011101	}	125	01111101
␞	30	00011110	>	62	00111110	^	94	01011110	~	126	01111110
␟	31	00011111	?	63	00111111	_	95	01011111	␡	127	01111111

STD-LL-60182

Katakana Display Characters

These characters can be displayed on all Series 200 Computers (while in Katakana mode).

ASCII	EQUIVALENT FORMS	
Char.	Dec	Binary
NOTE	128	10000000
NOTE	129	10000001
NOTE	130	10000010
NOTE	131	10000011
NOTE	132	10000100
NOTE	133	10000101
NOTE	134	10000110
NOTE	135	10000111
NOTE	136	10001000
NOTE	137	10001001
NOTE	138	10001010
NOTE	139	10001011
NOTE	140	10001100
NOTE	141	10001101
NOTE	142	10001110
NOTE	143	10001111
NOTE	144	10010000
NOTE	145	10010001
NOTE	146	10010010
NOTE	147	10010011
NOTE	148	10010100
NOTE	149	10010101
NOTE	150	10010110
NOTE	151	10010111
NOTE	152	10011000
NOTE	153	10011001
NOTE	154	10011010
NOTE	155	10011011
NOTE	156	10011100
NOTE	157	10011101
NOTE	158	10011110
NOTE	159	10011111

ASCII	EQUIVALENT FORMS	
Char.	Dec	Binary
	160	10100000
｡	161	10100001
｢	162	10100010
｣	163	10100011
､	164	10100100
･	165	10100101
ｦ	166	10100110
ｧ	167	10100111
ｨ	168	10101000
ｩ	169	10101001
ｪ	170	10101010
ｫ	171	10101011
ｬ	172	10101100
ｭ	173	10101101
ｮ	174	10101110
ｯ	175	10101111
ｰ	176	10110000
ｱ	177	10110001
ｲ	178	10110010
ｳ	179	10110011
ｴ	180	10110100
ｵ	181	10110101
ｶ	182	10110110
ｷ	183	10110111
ｸ	184	10111000
ｹ	185	10111001
ｺ	186	10111010
ｻ	187	10111011
ｼ	188	10111100
ｽ	189	10111101
ｾ	190	10111110
ｿ	191	10111111

ASCII	EQUIVALENT FORMS	
Char.	Dec	Binary
ﾀ	192	11000000
ﾁ	193	11000001
ﾂ	194	11000010
ﾃ	195	11000011
ﾄ	196	11000100
ﾅ	197	11000101
ﾆ	198	11000110
ﾇ	199	11000111
ﾈ	200	11001000
ﾉ	201	11001001
ﾊ	202	11001010
ﾋ	203	11001011
ﾌ	204	11001100
ﾍ	205	11001101
ﾎ	206	11001110
ﾏ	207	11001111
ﾐ	208	11010000
ﾑ	209	11010001
ﾒ	210	11010010
ﾓ	211	11010011
ﾔ	212	11010100
ﾕ	213	11010101
ﾖ	214	11010110
ﾗ	215	11010111
ﾘ	216	11011000
ﾙ	217	11011001
ﾚ	218	11011010
ﾛ	219	11011011
ﾜ	220	11011100
ﾝ	221	11011101
ﾞ	222	11011110
ﾟ	223	11011111

ASCII	EQUIVALENT FORMS	
Char.	Dec	Binary
	224	11100000
	225	11100001
	226	11100010
	227	11100011
	228	11100100
	229	11100101
	230	11100110
	231	11100111
	232	11101000
	233	11101001
	234	11101010
	235	11101011
	236	11101100
	237	11101101
	238	11101110
	239	11101111
	240	11110000
	241	11110001
	242	11110010
	243	11110011
	244	11110100
	245	11110101
	246	11110110
	247	11110111
	248	11111000
	249	11111001
	250	11111010
	251	11111011
	252	11111100
	253	11111101
	254	11111110
▓	255	11111111

NOTE: These are the same as the U.S./European characters.

STD-LL-60182

HP 9836 CRT Highlight Characters

ASCII char	halfbright	underline	blinking	inverse video
128				
129				X
130			X	
131			X	X
132		X		
133		X		X
134		X	X	
135		X	X	X
136	X			
137	X			X
138	X		X	
139	X		X	X
140	X	X		
141	X	X		X
142	X	X	X	
143	X	X	X	X

"X" means that this highlight is enabled by displaying the character.

Command Summaries

Main Command Level Summary

Assembler–Run the Assembler.

Compiler–Run the Pascal Compiler.

Debugger–Run the Debugger subsystem

Editor–Run the Editor subsystem.

eXecute–Execute a specified object file.

Filer–Run the Filer subsystem.

Initialize–Places all current blocked devices online.

Librarian–Run the Librarian subsystem.

Memvol–Create a memory resident volume.

Newsys–Select a new System Volume.

Permanent–Move an object file from a mass storage medium into internal read/write memory.

Run–Compile and execute the workfile or execute the last file compiled.

Stream–Stream a text file to be processed as keyboard commands.

User restart–Run the last program executed.

Version–Display version information about the Pascal Operating System.

What–Display the complete file specifier of each Pascal subsystem.

?–Display alternate command prompt.

Editor Command Summary

Text Modifying Commands

Copy – Insert text from the copy buffer or an external file in front of the current cursor location.

Delete – Remove text from the current cursor location to the location of the cursor when (Select) ((EXECUTE)) is pressed.

Insert – Inserts text in front of the current cursor location.

Replace – Replace the specified target string with the specified substitute string.

eXchange – Replace the text at the cursor with text typed from the keyboard, on a character-by-character basis.

Zap – Delete all text between the anchor and the current cursor location. (The anchor is set at the location of the latest Adjust, Find, Insert, or Replace command.)

Text Formatting Commands

Adjust – Adjust the column in which a line (or lines) start.

Margin – Format the paragraph the cursor is located to the margins in the current environment.

Miscellaneous Commands

Quit – Leave the Editor in an orderly manner. Provides various ways for saving the text currently in memory.

(STOP) ((SHIFT)-(CLR I/O)) – terminates the Editor subsystem, but the text is lost.

Set – Modify the environment or set markers in the text.

Verify – Update the displayed text to reflect the text stored in memory.

Cursor Keys

(TAB) – Move cursor to next tab position (fixed tabs) in the current direction.

(Return) or (ENTER) – Move cursor in current direction to the leftmost character in the next line.

Space Bar – Move cursor one character in the current direction.

Arrow Keys – Move cursor in the direction specified by the key.

Cursor Wheel – Moves the cursor like the arrow keys. Without (SHIFT), works like right and left arrows; with (SHIFT), works like the up and down arrows.

Cursor Positioning Commands

(=) – Typing (=) positions the cursor at the anchor. (The anchor is set at the location of the latest Adjust, Find, Insert, or Replace command.)

Find – Position the cursor after the specified target string.

Jump – Position the cursor at the beginning, the end, or the specified marker.

Page – Position the cursor ± 23 lines from the current location.

Filer Command Summary

Volume Related Commands

Bad sectors – Scans a volume and searches for unreliable (bad) storage areas.

Extended Directory – Lists directory information about a specified volume or set of files.

Krunch – Consolidates all unused space on a volume in a single area by packing the existing files together. (Not valid for SRM)

List Directory – Lists directory information about a specified volume or set of files.

Prefix – Specifies a new default volume.

Volumes – Lists the volumes currently on line.

Udir – Sets the default unit directory. (SRM only)

Zero – Creates an empty directory on the specified volume. (Not valid for SRM)

Exit Commands

Quit – Provides an orderly exit from the filer.

STOP – Pressing the (STOP) key unconditionally exits the Filer Subsystem. The current I/O operation is completed before exiting.

File Related Commands

Access – Change the access rights (passwords) on a file or directory. (SRM only)

Change – Change the name of a file, set of files, or volume.

Duplicate link – Duplicates links to a file or set of files. (SRM only)

Filecopy – Copies a file, set of files, or a volume to a specified destination.

Make – Create a directory (SRM) or a file on a volume.

Remove – Remove a directory entry or a set of directory entries.

Translate – Translates text files of types TEXT, ASCII, and DATA to other text file representations or to unblocked volumes.

Workfile Related Commands

Get – Specifies a file as the workfile.

New – Specifies that no file is the current workfile.

Save – Saves the current workfile(s) with the specified name.

What – Lists the name and current state (saved or not saved) of the workfile(s).

Librarian Command Summary

General Commands

Boot – creates "system Boot files."

Edit – gets you into Edit mode, for either Copying or Linking.

File – sends the File directory (all module names) to the current Printout file.

Header – allows you to specify the Header size for the Output file.

Input – allows you to specify the Input file.

Keep – makes a permanent copy of the current Output file.

Output – allows you to specify the Output file.

Printout – turns the Printout option ON or OFF, or allows you to specify a Printout file.

Quit – quits the Librarian and returns you to the Main Command Level.

Unassemble – gets you into the Unassemble mode.

Verify – gets you into the Verify mode, and shows the name of the first module in the Input.

Copy Mode Commands

All – transfers All modules from the Input file to the Output file.

Link – gets you into Link mode.

Module – allows you to specify the next Module to be copied from the Input file.

Transfer – Transfers the current object module to the Output file.

Edit Mode Commands

Append – Appends modules to the Output file.

Copy – Copies the First module up to (but not including) the Until module to the Output file.

First – allows you to specify the First module to be transferred to the Output file.

Stop – Stops the Edit session and returns to the Librarian's main prompt.

Transfer – Transfers the current object module to the Output file.

Until – allows you to specify the Until module.

Link Mode Commands

All – transfers All modules from the Input file to the Output file.

Copy – returns you to Copy mode.

Def – controls whether or not the DEF table is included in the Output file.

Global – allows you to change the Global base address of the module.

Link – finishes Linking.

Module – allows you to specify the next Module to be copied from the Input file.

New – allows you to name the New object module being created.

Relocation – designate the Relocation base address to be used.

Space – assigns Space for patches.

Transfer – Transfers the current object module to the Output file.

X – allows you to enter a copyright notice as part of the Output file.

Unassemble Commands

Assembler – directs the Librarian to unassemble the Input file using Assembler conventions.

Compiler – unassembles all lines of the Input file according to Compiler conventions.

Def – sends the DEF table to the Printout file.

Ext – sends the EXT table to the Printout file.

Line range – unassemble (using Compiler conventions) a section of code defined by two Line values.

PC range – unassemble (using Assembler conventions) a section of code defined by two location counter range values.

Stop – Stops the unassemble session and returns to the Librarian's main prompt.

Text – sends the interface Text (DEFINE SOURCE) of the current Input module to the Printout file.

Debugger Command Summary

Register Operations

A0..A7, D0..D7, PC, SP, US, SR – Display or assign values to the processor registers.

Breakpoint Commands

BS – Set a breakpoint at the specified location.

BD – Disable (but don't remove) breakpoints.

BA – Activate disabled breakpoints.

BC – Clear and remove breakpoints.

B – Display the breakpoint table

Call Command

CALL – Calls the machine-language routine at the specified address.

Display Command

D – Display the specified object(s) immediately, directly, or indirectly.

Dump Commands

DA – Performs a DUMP ALPHA function.

DG – Performs a DUMP ALPHA function.

Escape Commands

EC – Generates the specified escape code.

ET – Specify escape codes to be trapped by the Debugger.

ETC – Sets up trapping of all escape codes; the Debugger executes specified command(s) when an escape is encountered.

ETN – Specify that all escape codes except the ones listed are to be trapped by the Debugger.

Format Commands

FB – Sets the default display format to Binary.

FH – Sets default format to hex values.

FI – Sets default format to signed integer values.

FO – Sets the default display format to Octal.

FU – Sets default format to unsigned integer values.

Go Commands

G – Resume execution.

GT – Resume execution at a specified location.

GTF or **GFT** – Same as GT except execution is slowed and the line numbers are flashed in the lower right corner of the CRT.

IF, ELSE, and END Commands

IF, ELSE, END – Allow conditional execution of Debugger command(s).

Open Memory Commands

OB, OW, OL – Display or alter memory locations.

Procedure Commands

PN – Continues the program, but halts program execution when the next procedure is called (or current one is exited, whichever occurs first).

PX, or **P** – Continues the program, but halts program execution when the current procedure is exited.

Queue Commands

Q – List the most recent line numbers (or PC values if Trace commands were used with machine code).

QE – Terminate recording of line number values in the queue.

QS – Start recording the information in the queue.

Softkey Commands

k0 ... k9 – Define softkeys as typing-aid keys.

System Boot Command

sb – The system boot command puts the computer in the power-up state for rebooting.

Trace Commands

T – Execute the specified number of instructions, each followed by a TD command.

TD – Display the command string defined by the softkey k4.

TD I – Restores the initial command string to k4.

TQ – Same as the T command except the TD command is executed only after the last instruction.

TT – Same as the TQ command except a location is specified rather than a count.

Walk Procedure Links Commands

WD – Move the stack frame pointer to the stack frame of the calling procedure.

WS – Move the stack frame pointer to the stack frame of the nesting procedure.

WR – Return the stack frame pointer to the current stack frame.

Notes

Glossary

ASCII character Any of the 8-bit characters in Hewlett-Packard's extended ASCII (American Standard Code for Information Interchange) set. The characters include letters, numerals, punctuation, control characters and foreign character sets. A table of these characters and their code values can be found at the end of this glossary.

anchor An internal pointer used by the Editor's Zap command as a starting point for removing text. The anchor is set at the cursor position of the most recent Adjust, Find, Insert or Replace command. The cursor is moved to the anchor location by the Equals command.

bit An abbreviation for the term "binary digit", a bit is a single digit in base 2 that must be either a 0 or a 1.

block A block is a 512-byte unit of storage area on a WS1.0 volume and a 256-byte sector on a LIF volume. The Pascal system allocates storage space for files on the WS1.0 and LIF volumes in block increments.

block-structured An attribute of a device which structures its memory allocation in block units; examples are flexible or hard discs. Devices such as printers and screens (CRTs) are not block-structured.

boot device The peripheral where the Boot ROM found and loaded the Pascal operating system. The Boot ROM has a search pattern which allows booting from just about any drive in any HP mass storage product, including the Shared Resource Manager.

bus address When several peripherals are connected to the same HP-IB interface, a bus address is required (in addition to the select code) to designate the particular peripheral referenced by an I/O transaction.

byte A group of eight bits processed as a unit.

control character Any ASCII character whose value is either 127 or in the range of 0 thru 31. Use of control characters in the Editor and Filer is discouraged, because they may have undesirable effects.

cursor The flashing underline (_) symbol on the screen. The cursor functions as a reference point for Editor commands which manipulate text and as a reference for prompts in other Pascal subsystems.

cursor wheel The wheel (also called the knob) on the upper left area of the non-ITF keyboard whose action duplicates that of the four arrow keys. When used in the Editor with the (SHIFT) key pressed, turning the wheel produces up or down cursor movement; unshifted, it produces left or right cursor movement.

device selector By convention in the Pascal, BASIC and HPL systems, when select code and bus address are used together to address a peripheral, they are concatenated into a single number. Thus the device at address 1 on select code seven is referenced as 701, which is derived by multiplying the select code by 100 and adding the address. Some HP products contain, within a single package, several peripheral devices which must are addressed separately.

directory Contains information about the files on a volume. This information includes the volume name and the following information about each file on the medium: the file name, the file size (in number of blocks), the date of last modification to the file, its starting block address, and the file type (which reflects the file's attributes). Directory information can be seen by using the Filer's List Directory and Extended Directory commands. The directory is initialized with the Filer's Zero command.

Directory Access Method or DAM Each mass storage unit has a directory describing the files it contains, the type of each file and so forth. Many different directory organizations are used within HP, and data on a disc can't be interpreted properly unless it is accessed using the correct Directory Access Method. Pascal 2.0 supports three DAMs the "Workstation" format compatible with Pascal 1.0 systems; HP's "Logical Interchange Format" or LIF directory; and the Shared Resource Manager's hierarchical directory.

dollar sign This character "$" is used in the Filer as a convenience in specifying file names. When used in place of a destination file name, it means that the file is to have the same name as the source file.

entry point The place where a program or subroutine begins. Before a routine is executed, the address of the entry point must be obtained from a symbol table and that address is put in the program counter.

environment The conditions or parameters which affect how text in the Editor is Adjusted, Inserted, and Margined. These parameters may be changed with the Editor's Set command.

file A discrete collection of information designated by a file name and residing on a mass storage medium.

file name An entry in a directory which identifies a particular file.

file specification Completely identifies a file and may include both a volume specification and a file name. A volume specification can be one of many items, but it is always part of a file specification. If a volume ID is given, it must be separated from the file name by a colon (:). If not, the default volume is assumed.

file types Several file types are recognized by the Pascal System. Files generally (but not always) have a suffix as part of the file name which indicates their type. The file type is established at the time of the file's creation and cannot be changed just by changing the suffix. The types and their associated suffixes are:

- TEXT files - (suffix is .TEXT) Contain ASCII characters and Editor environment information.
- ASCII files - (suffix is .ASC) Are similar to TEXT files. The format is slightly different and there is no Editor environment information.
- CODE files - (suffix is .CODE) Contain code generated by the Pascal Assembler, Compiler or Librarian.
- Data files - (no specific suffix) Are files which can be created by any subsystem but are used primarily as INPUT and OUPUT files in Pascal programs. They do not have suffixes.
- System files - (suffix is .SYSTM) Are files created with the Librarian's Boot command. They are loadable by the boot ROM.
- Bad files - (suffix is .BAD) are a type of file created by the user to isolate unreliable or worn-out areas on a mass storage medium. Once created, BAD files will not be moved by subsequent crunches of the volume.

interface The electronic circuitry which connects the computer's high-speed internal bus to lower speed physical peripheral devices. Interfaces are either built-in, like the standard HP-IB port at the back of your computer, or plug into the I/O backplane. Most of the peripherals supported by the Series 200 computers are designed to connect through an HP-IB interface.

knob The rotary-pulse generator that is used as an input device on the built-in keyboards of the 9826 and 9836, and on optional HP 98203B keyboards. It is used in the Editor for moving the cursor (in that context, it is referred to as the "cursor wheel"). You can use it in programs for any purpose that you like.

Librarian A Pascal subsystem designed to manage object modules. It can link or just collect object modules together into object files. The Librarian is the file named LIBRARIAN in the operating system and is accessed by pressing L from the Main Command Level.

library A file that contains object modules. Libraries are the object of the compiler, Assembler, or Librarian.

LIBRARY A special library included with the Pascal operating system. This file is usually designated as the System Library at power-up. (The System Library is a special library file automatically accessed by the compiler and loader.) The System file should be kept on-line so that object modules stored in it are automatically available to any program importing them.

Main Command Level The level from which all the subsystems of the Pascal System are entered. The prompt displayed at this level looks like:

```
Command: Compiler Editor Filer Initialize Librarian Run eXecute Version ?
```

module See "object module".

mouse A small, rodent-like input device, consisting of a roller ball and buttons. Rolling the device on any surface generates two-dimensional movement information that is transmitted through its tail to the computer. Pushing the buttons also generates information that is sent to the computer. The mice available with Series 200 equipment are connected to the computer through the HP Human-Interface Link (HP-HIL).

object file An object file is a unit of binary code managed by the Librarian. It is made up of a Library directory and one or more object modules. The Assembler and Compiler generate one object file per source file. The Compiler's object file can contain one or more object modules depending upon the source file's construction. If the source file contains a number of compilable modules, that number of object modules will be created in the object file.

object module Contains the interface information necessary to link and run the module and the machine code.

on-line Any object (device, volume or file) currently accessible by the Pascal System.

opcode A word that stands for one of the operations of the microprocessor. The Assembler translates these words into actual binary codes which the microprocessor understands.

operand The symbol which stands for the object on which microprocessor operations are performed.

Pascal module HP Pascal allows program modules to be compiled separately into object modules. The modules are generally not executable, but are parts of Pascal programs. The sections of a module are:

```
MODULE, EXPORT, IMPORT, IMPLEMENT
```

pass by reference The address of a parameter variable is given to the called routine. Using that address, the routine can alter the value of the variable.

pass by value The current value of a variable is given to the called routine. In this way the value can be used but the routine does not alter the actual variable.

peripheral An I/O device such as a printer or disc.

prompt Generally, any request for information from the system. The different Pascal subsystems have primary prompts (the Editor Prompt, Filer Prompt, etc.) and many subsystem commands have prompts of their own which are displayed at the top of the screen when the command is entered.

pterodactyl A large flying reptile, presumed extinct.

relative addressing An addressing mode where the location of a routine or variable is given as an offset from the current location rather than an absolute address. In this way, the code can be placed at different places in memory without having to change the addresses of variables and entry points.

select code A number between 0 and 31, the "address" or name by which an interface is identified and referenced. When a peripheral operation is performed, it takes place through an interface which is said to be "on a select code". Most interface cards which plug into the I/O backplane have switches which can be set to indicate the select code to which the interface will respond. The built-in interfaces have fixed select codes.

string A contiguous series of non-control ASCII characters.

structured constant A constant that has more than a single value, such as a record or array.

structured variable A variable that has more than a single value, such as a record or array.

symbol table A table containing the address locations of the variables and routine entry points.

system volume or system unit The Pascal system distinguishes one mass storage unit to be used for special purposes. This "system volume" is where the date and any AUTOSTART file are found at boot time. It is where the system looks first for system files such as the Compiler and Editor, where workfiles are stored, and where an intermediate file is stored during interpretation of a Stream (command) file.

text file A file created and/or used by the Editor which contains ASCII or selected foreign characters. The Editor automatically appends `.TEXT` to a file name unless it either already contains a suffix or the last character in the file name is a period. A text file may be of type TEXT, ASCII or DATA.

unit An entry in the Unit Table.

Unit Table The Pascal system provides for up to 50 units, designated #1 through #50. They are represented by a 50-entry array called the Unit Table or "Unitable". Each entry fully specifies the association of one logical unit to a physical peripheral, with such information as the device selector and driver procedures to be used for I/O operations to the unit.

unit number An integer in the range from 1 through 50 representing the volume having the corresponding entry in the unit table.

volume A volume refers to any I/O device such as a printer, keyboard, screen, or mass storage device. The name of a mass storage volume is found in its directory; the name of an unblocked device is found in its Unit Table entry. There may be several volumes on one physical storage medium. Hard discs typically contain multiple volumes, but flexible discs generally have only a single volume. The volume may be mounted (in a disc drive) or not. The syntax of a volume name depends on its type (for example, LIF volume names may contain 6 characters, WS1.0 may contain 7).

wildcard Both of the characters = and ? can be used in the Filer as wildcards in place of parts of a file specification.

workfile If the workfile exists, it is the automatic file used by the Editor, Compiler, Assembler, Debugger and the Run command. It is designated when quitting the Editor using the Update option or the Filer's Get command.

Error Messages

This appendix contains all of the error messages and conditions that you are likely to encounter while using the Pascal system. They can be placed into the following categories; each category is discussed in a subsequent section.

- Unreported errors – certain errors do not get reported by this implementation of Pascal.
- Boot-time errors – These are errors that occur while the Pascal system is booting (they are reported by the system loader).
- Run-time errors – These are general errors which may occur while you are using the system. Run-time errors −10, −26, and −27 have special meanings, as described below.
- I/O System errors – When run-time error −10 occurs, there has been a problem with the I/O system. The operating system then prints an error message from the list of I/O system errors.
- I/O Library errors – When run time error −26 occurs, there has been a problem in an `IO` library procedure.
- Graphics Library errors – When run time error −27 occurs, there has been a problem in a `GRAPHICS` library procedure.
- Compiler syntax errors.
- Editor, Filer, and Debugger errors and conditions.

Unreported Errors

The following errors in Pascal programs are not reported by this implementation of the language.

- Disposing a pointer while in the scope of a WITH referencing the variable to which it points.
- Disposing a pointer while the variable it points to is being used as a VAR parameter.
- Disposing an uninitialized or NIL pointer.
- Disposing a pointer to a variant record using the wrong tagfield list.
- Assignment to a FOR-loop control variable while inside the loop.
- GOTO into a conditional or structured statement.
- Exiting a function before a result value has been assigned.
- Changing the tagfield of a dynamic variable to a value other than was specified in the call to NEW.
- Accessing a variant field when the tagfield indicates a different variant.
- Negative field width parameters in a WRITE statement.
- The underscore character "_" is allowed in identifiers. This is permitted in HP Pascal, but is not reported as an error when compiling with `$ANSI$` specified.
- Value range error is not always reported when an illegal value is assigned to a variable of type SET.

Boot-Time Errors

Errors that occur while your system is booting will report a message like this:

```
    IORESULT, ERROR: 0, 112
```

The value of IORESULT is shown first (0 in the above display). See the I/O System Errors section for descriptions of those error numbers.

The value of ERROR is shown second (112 in the above display). See the Loader/SEGMENTER Errors section for a description of those error numbers.

Run-Time Errors

Errors detected by the operating system during the execution of a program generate one of the error messages listed on this page (unless you trap it with a TRY..RECOVER construct).

Note

Note that error −10 occurs, the error message listed here will *not* be shown; the message on the next page (in I/O System Errors) will be shown instead.

When using a TRY..RECOVER construct (which requires the $SYSPROG ON$ Compiler option), the following numbers correspond to the value returned by the ESCAPECODE function.

0 Normal termination.

− **1** Abnormal termination.

− **2** Not enough memory.

− **3** Reference to NIL pointer.

− **4** Integer overflow.

− **5** Divide by zero.

− **6** Real math overflow. (The number was too large.)

− **7** Real math underflow. (The number was too small.)

− **8** Value range error.

− **9** Case value range error.

− **10** Non-zero IORESULT. (Note that the corresponding message on the next page, not this message, will be shown.)

− **11** CPU word access to odd address.

− **12** CPU bus error.

− **13** Illegal CPU instruction.

− **14** CPU privilege violation.

− **15** Bad argument - SIN/COS.

− **16** Bad argument - Natural Log.

− **17** Bad argument - SQRT. (Square root.)

− **18** Bad argument - real/BCD conversion.

− **19** Bad argument - BCD/real conversion.

− **20** Stopped by user.

− **21** Unassigned CPU trap.

− **22** Reserved

− **23** Reserved

− **24** Macro Parameter not 0..9 or a..z.

− **25** Undefined Macro parameter.

− **26** Error in I/O subsystem.

− **27** Graphics routine error.

− **28** Parity error in memory.

− **29** Misc. floating-point hardware error.

I/O System Errors

These error messages are automatically printed by the system unless you have enclosed the error-producing statement in a TRY..RECOVER construct. Within the RECOVER block, the ESCAPECODE function returning a value of −10 indicates that one of the following errors has occurred; you can determine which error has occurred by using the IORESULT function.

0 No I/O error reported.

1 Parity (CRC) incorrect.

2 Illegal unit number.

3 Illegal I/O request.

4 Device timeout.

5 Volume went off-line.

6 File lost in directory.

7 Bad file name.

8 No room on volume.

9 Volume not found.

10 File not found.

11 Duplicate directory entry.

12 File already open.

13 File not open.

14 Bad input format.

15 Disc block out of range.

16 Device absent or unaccessible.

17 Media initialization failed.

18 Media is write protected.

19 Unexpected interrupt.

20 Hardware/media failure.

21 Unrecognized error state.

22 DMA absent or unavailable.

23 File size not compatible with type.

24 File not opened for reading.

25 File not opened for writing.

26 File not opened for direct access.

27 No room in directory.

28 String subscript out of range.

29 Bad file close string parameter.

30 Attempt to read or write past end-of-file mark.

31 Media not initialized.

32 Block not found.

33 Device not ready or medium absent.

34 Media absent.

35 No directory on volume.

36 File type illegal or does not match request.

37 Parameter illegal or out of range.

38 File cannot be extended.

39 Undefined operation for file.

40 File not lockable.

41 File already locked.

42 File not locked.

43 Directory not empty.

44 Too many files open on device.

45 Access to file not allowed.

46 Invalid password.

47 File is not a directory.

48 Operation not allowed on directory.

49 Cannot create /WORKSTATIONS/TEMP_FILES.

50 Unrecognized SRM error.

51 Medium may have been changed.

52 IO result was 52.

I/O Library Errors

When run-time error −26 occurs, there has been a problem in an I/O library procedure. By importing the IODECLARATIONS module, you can use the IOE_RESULT and IOERROR_MESSAGE functions to get a textual error description. For example:

```
$SYSPROG ON$
      ...
import IODECLARATIONS, GENERAL_3
      ...
begin
try
      ...
recover
    if ESCAPECODE = IOESCAPECODE
        then writeln (IOERROR_MESSAGE(IOE_RESULT));
    ESCAPE(ESCAPECODE);
end.
```

IOESCAPECODE is a constant (= −26) which you can import from the IODECLARATIONS module. ESCAPE is a procedure and ESCAPECODE is a function; both are accessible when you use the $SYSPROG ON$ Compiler option.

0	No error.
1	No card at select code.
2	Interface should be HP-IB.
3	Not active controller.
4	Should be device address, not select code.
5	No space left in buffer.
6	No data left in buffer.
7	Improper transfer attempted.
8	The select code is busy.
9	The buffer is busy.
10	Improper transfer count.
11	Bad timeout value.
12	No driver for this card.
13	No DMA.
14	Word operations not allowed.
15	Not addressed as talker.
16	Not addressed as listener.
17	A timeout has occurred.
18	Not system controller.
19	Bad status or control.
20	Bad set/clear/test operations.
21	Interface card is dead.
22	End/eod has occurred.
23	Miscellaneous - value of parameter error.
306	Data-Comm interface failure.
313	USART receive buffer overflow.
314	Receive buffer overflow.
315	Missing clock.
316	CTS false too long.
317	Lost carrier disconnect.
318	No activity disconnect.
319	Connection not established.
325	Bad data bits/parity combination.
326	Bad status/control register.
327	Control value out of range.

Graphics Errors

When run-time error −27 occurs, there has been an error in a GRAPHICS library routine.

By importing the DGL_LIB module, you can call the GRAPHICSERROR function which returns an INTEGER value you can cross reference with the numbered list of graphics errors.

```
$SYSPROG ON$
      ...
import DGL_LIB;
      ...
begin
try
      ...
recover
    if ESCAPECODE = -27
        then writeln ('Graphics error #', GRAPHICSERROR,
                      ' has occurred')
        else ESCAPE(ESCAPECODE);
end.
```

You may wish to write a procedure which takes the INTEGER value from GRAPHICSERROR and prints the description of the error on the CRT. You could keep this procedure with your program or, for more global use, in the System Library (normally SYSVOL:LIBRARY).

0 No error. {Since last call to graphicserror or into_graphics.}

1 The graphics system is not initialized.

2 The graphics display is not enabled.

3 The locator device is not enabled.

4 ECHO value requires a graphic display to be enabled.

5 The graphics system is already enabled.

6 Illegal aspect ratio specified.

7 Illegal parameters specified.

8 The parameters specified are outside the physical display limits.

9 The parameters specified are outside the limits of the window.

10 The logical locator and the logical display use the same physical device. {The logical locator limits cannot be redefined explicitly. They must correspond to the logical view surface limits.}

11 The parameters specified are outside the current virtual coordinate system boundary.

12 The escape function requested is not supported by the graphics display device.

13 The parameters specified are outside of the physical locator limits.

Loader/SEGMENTER Errors

Here is a list of errors that can be generated by a program that uses the SEGMENTER module (or by the loader; see Boot-Time Errors):

Error	Meaning
100..105	field overflow trying to link or relocate something
110	circular or too deeply nested symbol definitions
111	improper link info format
112	not enough memory
116	file was not a code file
117	not enough space in the explicit global area
118	incorrect version number
119	unresolved external references
120	generated by the dummy procedure returned by find_proc
121	unload_segment called when there are no more segments to unload
122	not enough space in the explicit code area

SEGMENTER Errors

When one of these errors occurs while using the SEGMENTER module procedures, you can determine which has occurred by using a TRY..RECOVER construct and calling the ESCAPECODE function in the RECOVER block.

Loader Boot-Time Errors

When an error occurs while booting, a message such as the following will be reported:

```
IORESULT, ERROR =     0,  112
```

The second number indicates which loader error has occurred. (The first number indicates which I/O system error has occurred; see the preceding I/O System Errors section for descriptions of each error.)

Pascal Compiler Errors

The following errors may occur during the compilation of a HP Pascal program.

ANSI/ISO Pascal Errors

1 Erroneous declaration of simple type

2 Expected an identifier

4 Expected a right parenthesis “)”

5 Expected a colon “:”

6 Symbol is not valid in this context

7 Error in parameter list

8 Expected the keyword OF

9 Expected a left parenthesis “(“

10 Erroneous type declaration

11 Expected a left bracket “[”

12 Expected a right bracket “]”

13 Expected the keyword END

14 Expected a semicolon “;”

15 Expected an integer

16 Expected an equal sign “=”

17 Expected the keyword BEGIN

18 Expected a digit following “.”

19 Error in field list of a record declaration

20 Expected a comma “,”

21 Expected a period “.”

22 Expected a range specification symbol “..”

23 Expected an end of comment delimiter

24 Expected a dollar sign “$”

50 Error in constant specification

51 Expected an assignment operator “:=”

52 Expected the keyword THEN

53 Expected the keyword UNTIL

54 Expected the keyword DO

55 Expected the keyword TO or DOWNTO

56 Variable expected

58 Erroneous factor in expression

59 Erroneous symbol following a variable

98 Illegal character in source text

99 End of source text reached before end of program

100 End of program reached before end of source text

101 Identifier was already declared

102 Low bound > high bound in range of constants

103 Identifier is not of the appropriate class

104 Identifier was not declared

105 Non-numeric expressions cannot be signed

106 Expected a numeric constant here

107 Endpoint values of range must be compatible and ordinal

108 NIL may not be redeclared

110 Tagfield type in a variant record is not ordinal

111 Variant case label is not compatible with tagfield

113 Array dimension type is not ordinal

115 Set base type is not ordinal

117 An unsatisfied forward reference remains

121 Pass by value parameter cannot be type FILE

123 Type of function result is missing from declaration

125 Erroneous type of argument for built-in routine

126 Number of arguments different from number of formal parameters

127 Argument is not compatible with corresponding parameter

129 Operands in expression are not compatible

130 Second operand of IN is not a set

131 Only equality tests (=, <>) allowed on this type

132 Tests for strict inclusion (<, >) not allowed on sets

133 Relational comparison not allowed on this type

134 Operand(s) are not proper type for this operation

135 Expression does not evaluate to a boolean result

136 Set elements are not of ordinal type

137 Set elements are not compatible with set base type

138 Variable is not an ARRAY structure

139 Array index is not compatible with declared subscript

140 Variable is not a RECORD structure

141 Variable is not a pointer or FILE structure

143 FOR loop control variable is not of ordinal type

144 CASE selector is not of ordinal type

145 Limit values not compatible with loop control variable

147 Case label is not compatible with selector

149 Array dimension is not bounded

150 Illegal to assign value to built-in function identifier

152 No field of that name in the pertinent record

154 Illegal argument to match pass by reference parameter

156 Case label has already been used

158 Structure is not a variant record

160 Previous declaration was not forward

163 Statement label not in range 0..9999

164 Target of nonlocal GOTO not in outermost compound statement

165 Statement label has already been used

166 Statement label was already declared

167 Statement label was not declared

168 Undefined statement label

169 Set base type is not bounded

171 Parameter list conflicts with forward declaration

177 Cannot assign value to function outside its body

181 Function must contain assignment to function result

182 Set element is not in range of set base type

183 File has illegal element type

184 File parameter must be of type TEXT

185 Undeclared external file or no file parameter

190 Attempt to use type identifier in its own declaration

300 Division by zero

301 Overflow in constant expression

302 Index expression out of bounds

303 Value out of range

304 Element expression out of range

400 Unable to open list file

401 File or volume not found

403.. 409 Compiler error

Compiler options

600 Directive is not at beginning of the program

601 Indentation too large for $PAGEWIDTH

602 Directive not valid in executable code

604 Too many parameters to $SEARCH

605 Conditional compilation directives out of order

606 Feature not in Standard PASCAL flagged by $ANSI ON

607 Feature only allowed when $UCSD enabled

608 $INCLUDE exceeds maximum allowed depth of files

609 Cannot access this $INCLUDE file

610 $INCLUDE or IMPORT nesting too deep to IMPORT <module-name>

611 Error in accessing library file

612 Language extension not enabled

613 Imported module does not have interface text

614 LINENUM must be in the range 0..65535

620 Only first instance of routine may have $ALIAS

621 $ALIAS not in procedure or function header

646 Directive not allowed in EXPORT section

647 Illegal file name

648 Illegal operand in compiler directive

649 Unrecognized compiler directive

Implementation restrictions

651 Reference to a standard routine that is not implemented

652 Illegal assignment or CALL involving a standard procedure

653 Routine cannot be followed by CONST, TYPE, VAR, or MODULE

655 Record or array constructor not allowed in executable statement

657 Loop control variable must be local variable

658 Sets are restricted to the ordinal range 0 .. 255

659 Cannot blank pad literal to more than 255 characters

660 String constant cannot extend past text line

661 Integer constant exceeds the range implemented

662 Nesting level of identifier scopes exceeds maximum (20)

663 Nesting level of declared routines exceeds maximum (15)

665 CASE statement must contain a non-OTHERWISE clause

667 Routine was already declared forward

668 Forward routine may not be external

671 Procedure too long

672 Structure is too large to be allocated

673 File component size must be in range 1..32766

674 Field in record constructor improper or missing

675 Array element too large

676 Structured constant has been discarded (cf. $SAVE_CONST)

677 Constant overflow

678 Allowable string length is 1..255 characters

679 Range of case labels too large

680 Real constant has too many digits

681 Real number not allowed

682 Error in structured constant

683 More than 32767 bytes of data

684 Expression too complex

685 Variable in READ or WRITE list exceeds 32767 bytes

686 Field width parameter must be in range 0..255

687 Cannot IMPORT module name in its EXPORT section

688 Structured constant not allowed in FORWARD module

689 Module name may not exceed 15 characters

696 Array elements are not packed

697 Array lower bound is too large

698 File parameter required

699 32-bit arithmetic overflow

Non-ISO Language Features

701 Cannot dereference (^) variable of type anyptr

702 Cannot make an assignment to this type of variable

704 Illegal use of module name

705 Too many concrete modules

706 Concrete or external instance required

707 Variable is of type not allowed in variant records

708 Integer following # is greater than 255

709 Illegal character in a "sharp" string

710 Illegal item in EXPORT section

711 Expected the keyword IMPLEMENT

712 Expected the keyword RECOVER

714 Expected the keyword EXPORT

715 Expected the keyword MODULE

716 Structured constant has erroneous type

717 Illegal item in IMPORT section

718 CALL to other than a procedural variable

719 Module already implemented (duplicate concrete module)

720 Concrete module not allowed here

730 Structured constant component incompatible with corresponding type

731 Array constant has incorrect number of elements

732 Length specification required

733 Type identifier required

750 Error in constant expression

751 Function result type must be assignable

900 Insufficient space to open code file

901 Insufficient space to open ref file

902 Insufficient space to open def file

903 Error in opening code file

904 Error in opening ref file

905 Error in opening def file

906 Code file full

907 Ref file full

908 Def file full

Assembler Errors

Error messages are listed under the line in which they occur. At the completion of the assembly, the number of errors will be displayed. If there are errors, there will be a directive for you to check the location of the last error in the program. At that location there will be a description of the error. Also listed will be the location of the error above it if one exists. In this manner, all errors can be located without having to search the whole listing.

Error Messages

Address Register Expected.

Attempt to Nest Included Files.

Blank orEOL Expected.

Comma Expected.

Code Segment Starts at Odd Address.

Duplicate Definition of Symbol.

Error Reading Code File.

Error Reading Source File.

Error Writing Code File.

Error Writing Source File.

Error Reading Code File.

Expression is Improper Mode.

External Reference Not Allowed.

Failed to Open Included File. File could not be found.

Field Overflow. A specification of the assembly instruction will not fit within the appropriate field of the machine instruction.

Illegal Constant.

Illegal Expression.

Illegal Operand Size for this Instruction.

Illegal Syntax.

Improper Addressing Mode

Improper Use of Mode Declaration. Symbol already has mode or declaration appears after first use of symbol.

Improper Use of Size Suffix.

Invalid Opcode.

Label Required. For some pseudo-ops.

Module Directory Overflow. Too many ORG or RORG statements in assembly or Non-Contiguous DS statements.

More Than One COM Statement.

More Than One MNAME Statement.

More Than One START Statement.

Phase Error. A symbol value is not equal in both passes.

Register or Register List Expected.

Right Parenthesis Expected.

Symbol Expected.

Undefined Symbol.

Debugger Error Messages/Conditions

WHAT?
The first characters of a command are not recognized.

SYNTAX ERROR
Somewhere in the current command, the syntax rules for the command have been violated.

OVERFLOW
A number entered or the result of an arithmetic operation cannot be represented in 32 bits.

BUSERROR
An address has been accessed which does not exist in the machines configuration.

INPUT OVERFLOW
An internal input stack has overflowed.

ADDRESS ERROR
An old address has been referenced when an even address is required.

TOO MANY CODES
Too many escape codes in the ET or ETN list.

SIZE ERROR
An entered value does not fit in the required space e.g., registers.

TYPE ERROR
The parameter entered for a command is not the correct type. e.g., an alpha value when a line number or address is required.

EXPRESSION TOO COMPLEX
The expression requires too much stack space to execute. e.g., more than three levels of parentheses.

DIVIDE BY ZERO
The value to the right of the / symbol is zero.

UNDEFINED SYMBOL
An expression contains a reference to a symbol which the DEBUGGER does not recognize.

SIZE FIELD TOO BIG
In a format, the size field is too large for the object being dumped or the format spec being used e.g., the size field for I and U is 1..4. The default size for string data is the length of the string.

FORMAT REQUIRES MORE DATA
An attempt has been made to display more bytes than the object contains.

ADDRESS FORMAT NOT ALLOWED
The *, < > and ^ format codes are only allowed if the object is type address.

PC/SP HAS ODD ADDRESS
An attempt to return to the user code has been made under the above conditions.

DUPLICATE BREAK
GT orTT has specified a location which already has a break point defined.

Subject Index

Manual Comment Sheet Instruction

If you have any comments or questions regarding this manual, write them on the enclosed comment sheets and place them in the mail. Include page numbers with your comments wherever possible.

If there is a revision number, (found on the Printing History page), include it on the comment sheet. Also include a return address so that we can respond as soon as possible.

The sheets are designed to be folded into thirds along the dotted lines and taped closed. Do not use staples.

Thank you for your time and interest.

MANUAL COMMENT SHEET

Pascal 3.0 Workstation System
for the HP 9000 Series 200

98615-90021

May 1984

Update No. ____________

(See the Printing History in the front of the manual)

Name: ____

Company: ____

Address: ____

____

Phone No: ____________________

fold -- fold

Programming Experience: ______________________________

____

System Configuration: ______________________________

____

____

Comments: ______________________________

____

____

____

____

____

fold -- fold

____

____

____

____

____

____

____

____

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HEWLETT
PACKARD

5957-2657 Printed in U.S.A. Revised December 1984